NASA Conference Publication 10162 Vol. II

Third CLIPS Conference Proceedings

Gary Riley, Editor
Lyndon B. Johnson Space Center
Houston, Texas

Proceedings of a conference sponsored by
Lyndon B. Johnson Space Center and I-NET, Inc.
and held at the Johnson Space Center, Houston, Texas
September 12-14, 1994

1994

This publication is available from the NASA Center for AeroSpace Information,
800 Elkridge Landing Road, Linthicum Heights, MD 21090-2934, (301) 621-0390.

ABSTRACT

Expert systems are computer programs which emulate human expertise in well defined problem domains. The potential payoff from expert systems is high: valuable expertise can be be captured and preserved, repetitive and/or mundane tasks requiring human expertise can be automated, and uniformity can be applied in decision making processes. The C Language Integrated Production System (CLIPS) is an expert system building tool, developed at the Johnson Space Center, which provides a complete environment for the development and delivery of rule and/or object based expert systems. CLIPS was specifically designed to provide a low cost option for developing and deploying expert system applications across a wide range of hardware platforms. The development of CLIPS has helped to improve the ability to deliver expert system technology throughout the public and private sectors for a wide range of applications and diverse computing environments. The Third Conference on CLIPS provided a forum for CLIPS users to present and discuss papers relating to CLIPS applications, uses, and extensions.

CONTENTS

VOLUME I

VOLUME II

omit

Session 4B: Parallel/Distributed Processing Extensions

Session Chair: Len Myers

N95-19748 -61
34084
P.9

USING PVM TO HOST CLIPS IN DISTRIBUTED ENVIRONMENTS

Leonard Myers
Computer Science Department
California Polytechnic State University
San Luis Obispo, CA 93402
lmyers@calpoly.edu

Kym Pohl
CAD Research Center
California Polytechnic State University
San Luis Obispo, CA 93402
kpohl@calpoly.edu

Abstract

It is relatively easy to enhance CLIPS to support multiple expert systems running in a distributed environment with heterogeneous machines. The task is minimized by using the PVM (Parallel Virtual Machine) code from Oak Ridge Labs to provide the distributed utility. PVM is a library of C and FORTRAN subprograms that supports distributive computing on many different UNIX platforms. A PVM deamon is easily installed on each CPU that enters the virtual machine environment. Any user with *rsh* or *rexec* access to a machine can use the one PVM deamon to obtain a generous set of distributed facilities. The ready availability of both CLIPS and PVM makes the combination of software particularly attractive for budget conscious experimentation of heterogeneous distributive computing with multiple CLIPS executables. This paper presents a design that is sufficient to provide essential message passing functions in CLIPS and enable the full range of PVM facilities.

Introduction

Distributed computing systems can provide advantages over single CPU systems in several distinct ways. They may be implemented to provide one or any combination of the following: parallelism, fault tolerance, heterogeneity, and cost effectiveness. Our interest in distributive systems is certainly for the cost effectiveness of the parallelism they provide. The speedup and economy possible with multiple inexpensive CPUs executing simultaneously make possible the applications in which we are interested, without a supercomputer host. The economy of distributed computing may be necessary for the realization of our applications, but we are even more interested in distributed systems to provide a simplicity of process scheduling and rapid interaction of low granularity domain specific actions. Our approach to implementing certain large and complex computer systems is focused on the notion of quickly providing reasoned response to user actions.

We accomplish the response by simultaneously examining the implications of an user action from many points of view. Most often these points of view are generated by individual CLIPS[1] expert systems. However, there is not necessarily a consistency in these viewpoints. Also, a domain inference may change over time, even a small period of time, as more information becomes available. Our systems have the experts immediately receive and respond to low level user action representations. The action representations encode the lowest level activities that are meaningful to the user, such as the drawing of a line in a CAD environment. Thus, if there are a dozen, or more, expert domains involved in an application, it is necessary to transmit user actions to each expert and combine the responses into a collective result that can be presented to the user. Of course, the intention is to emulate, in a relatively gross manner, the asynchronous parallel cooperative manner in which a biological brain works.

Over the course of eight years, the CAL POLY CAD Research Center and CDM Technologies, Incorporated have developed several large applications and many small experiments based essentially on the same ideas.[2, 3, 4] In the beginning we wrote our own socket code to communicate between the processes. Unfortunately, with each new brand of CPU introduced into

a system it was necessary to modify the communication code. There simply are not sufficient standards in this area to provide portability of code, except by writing functions that are machine specific. Moreover, our main interest is not the support of what is essentially maintenance of the inter-process communication code. But for many years it seemed necessary to support our own communication code because the other distributed systems we investigated were simply too inefficient for our applications, even though they were generally much more robust. We then found that PVM[5] code was very compatible with our requirements.

PVM

PVM is a free package of programs and functions that support configuration, initiation, monitoring, communication, and termination of distributed UNIX processes among heterogeneous CPUs. Version 3.3 makes UDP sockets available for communication between processes on the same CPU, which is about twice as fast as the TCP sockets generally necessary for communication between CPUs. Also, shared memory communication is possible on SPARC, SGI, DEC, and IBM workstations, and several multiprocessor machines.

Any user can install the basic PVM daemon on a CPU and make it available to all other users. It does not require any root privileges. Users can execute their own console program, *pvm3*, that provides basic commands to dynamically configure the set CPUs that will make up the 'virtual machine'. The basic daemon, *pvmd*, will establish unique dynamic socket communications for each user's virtual machine. Portability is obtained through the use of multiple functions that are written specifically for each CPU that is currently supported. An environment variable is used to identify the CPU type of each daemon and hardware specific directory names, such as *SUN4*, are used to provide automatic recognition of the proper version of a PVM function. Since all of the common UNIX platforms are currently supported, the code is very portable.

The fundamental requirements for a PVM distributed system involving CLIPS processes consist of starting a CLIPS executable on each selected CPU and calling PVM communication facilities from CLIPS rulesets. The first of these tasks is very easy. PVM supports a 'spawn' command from the console program as well as a 'pvm_spawn' function that can be called from a user application program. Both allow the user to let PVM select the CPU on which each program will be executed or permit the user the specify a particular CPU or CPU architecture to host a program. Over sixty basic functions are supported for application programming in C, C++, or FORTRAN, but only the most basic will be mentioned here and only in their C interface forms.

PVM Communication Basics

Each PVM task is identified by its 'task id'(tid), an integer assigned by the local PVM daemon. The tid is uniquely generated within the virtual machine when either the process is spawned, or the pvm_mytid function is first called. Communication of messages between two processes in a PVM system generally consists of four actions:

1. clear the message buffer and establish a new one - **pvm_initsend**
2. pack typed information into the message buffer - **pvm_pkint, pvm_pkfloat,** etc.
3. send the message to a PVM task or group of tasks - **pvm_send, pvm_mcast,** etc.
4. receive the message, either blocking or not - **pvm_recv, pvm_nrecv,** etc.

Each message is required to be assigned an integer, 'msgtype', which is intended to identify its format. For instance, task1 may send a message that consists of ten integers to task2. In the

'pvm_send' used to transmit the message, the last argument is used to identify the type of message. The programmer may decide that such a message is to be of type 4, for example. Then, task2 can execute 'pvm_recv', specifying which task and message type it would like to receive. A -1 value can be used as a wildcard for either parameter in order to accept messages from any task and/or of any type. If task2 indicates that it wishes to accept only a message of type 4 to satisfy this call, it then knows when the 'pvm_recv' blocking function succeeds that it has a message of ten integers in a new input buffer.

As a more complete example, the following code shows a host task that will spawn a client task; then it will send the client a message consisting of an integer and a floating point value; and then it will wait for a reply from the client. The client waits for the message from the host, generates a string message in reply, and halts. When the client message is received, the host prints the message and halts.

Host

```
#include "pvm3.h"
main () {

int hostid, tids[1];/* pvm encodes pid within a taskid (tid) */
int ivalue1;       /* first value to be sent                  */
float fvalue2;     /* second value to be sent                 */
char message[25];/* string to be returned from client        */

hostid = pvm_mytid;              /* enroll in pvm */

pvm_spawn("Client",      /* name of process to start up  */
          (char**)0,     /* args to be passed to process */
          0,             /* options - 0 to use any CPU   */
          "",            /* host name when not option 0*/
          1,             /* number of copies to spawn    */
          tids);         /* tids of processes started    */

pvm_initsend(PvmDataRaw); /* get buffer                   */
                          /* no encoding of data if same type CPU */
pvm_pkint(23, 1, 1);      /* pack 1 int into the current buffer   */
pvm_pkfloat(45.678, 1, 1); /* pack 1 float into the current buffer */
pvm_send(tid[0], 2);      /* send buffer as user chosen type 2    */

pvm_recv(-1, 1);          /* wait for message from anyone (-1 is a */
                          /* wildcard) that is user typed as 1     */
pvm_upkstr(message);      /* unpack string from buffer into message*/

printf("I got the message: %s\n", message);

pvm_exit();
}
```

Client

```
#include "pvm3.h"
main() {

int hostid, clientid;
clientid = pvm_mytid;

pvm_recv(-1, 2);       /* wait for type 2  */
/* we ignore the contents in this example */

pvm_initsend(PvmDataRaw);
pvm_pkstr("Hi Host!"); /* pack string      */
hostid = pvm_parent();     /* get host tid */
pvm_send(hostid, 1);       /* send string  */
                           /* as type 1    */

pvm_exit();
}
```

Figure 1. A PVM Example

CLIPS Implementation Considerations

The basic need is to assert a fact or template from a rule in one CLIPS process into the factlist of another CLIPS process, which may be executing on a different processor. We might want a function that could use a PVM buffer to transmit a CLIPS 'person' template as in figure 2:

```
( BMPutTmplt ?BufRef ( person  (name  "Len")  ( hair "brown") (type "grumpy")) )
```

Figure 2. Sample Send Template Call

The first problem this presents is that the form of this function call is illegal. It is possible to use a syntax in which the fact that is to be communicated is presented as a sequence of arguments, rather than a parenthesized list. But this syntax does not preserve the appearance of the local assert, which is pleasing to do. The solution is to add some CLIPS source code to provide a parser for this syntax.

Second, consideration must be given as to when the PVM code that receives the messages should execute. It is desirable that receiving functions can be called directly from the RHS of a CLIPS rule. It is also desirable in most of our applications that message templates are transparently asserted and deleted from the receiver's factlist without any CLIPS rules having to fire. In order to accommodate the latter, our CLIPS shell checks for messages after the execution of every CLIPS rule, and blocks for messages in the case that the CLIPS agenda becomes empty.

Third, it is useful to be able to queue message facts to be asserted and have them locally inserted into the receiver's factlist during the same CLIPS execution cycle. There are occasions when a number of rules might fire on a subset of the message facts that are sent. In most cases the best decision as to what rule should fire can be made if all associated message facts are received during one cycle, and all such rules are activated at the same time. Salience can usually be used to select the best of such rules and the execution of the best rule can then deactivate the other alternatives. In order to implement this third consideration, the message facts are not sent for external assertion until a special command is given.

OVERVIEW OF CMS

The CLIPS/PVM interface or CLIPS Message System (CMS), provides efficient, cost effective communication of simple and templated facts among a dynamic set of potentially distributed and heterogeneous clients. These clients may be C, C++, or CLIPS processes. Clients may communicate either single fact objects or collections of facts as a single message. This communication takes place without requiring either the sender or receiver to be aware of the physical location or implementation language of the other. CMS will transmit explicit generic simple fact forms by dynamically building C representations of any combination of basic CLIPS atoms, such as INTEGERs, FLOATs, STRINGs, and MULTIFIELDs. CMS will also communicate facts or templates referenced by a fact address variable.

In addition to the dynamic generic process described above, the communication of templated facts is supported in a more static and execution time efficient manner. Specific routines capable of manipulating client templates in a direct fashion are written for each template that is unique in the number and type of its fields The deftemplate identifies what message form these routines will expect. This is a distinct luxury that eliminates the time involved in dynamically determining the number and types of fields for a generic fact that is to be communicated. The disadvantage is that the routines must be predefined to match the templates to be communicated. However, the applications in which the authors are involved have predetermined vocabularies for the data objects that are to be communicated. In much the same way that deftemplates provide efficiency through the use of predetermined objects, these template-form specific communication routines provide object communication in the most efficient, cost-effective fashion. The following section describes the overall architecture of a system which supports the previously described functionality.

Architecture

The underlying support environment consists of a set of utility managers called the System Manager, Memory Manager, and Error Manager. All three managers provide various

functionality utilized by the communication system to perform efficient memory usage and error notification. The core of the communication system consists of five inter-related components as shown in Figure 3. These components are the Session Manager (SM), Buffer Manager (BM), Communication Object Class Library, Communication Object META Class, and CLIPS Interface Module. Collectively, these components provide the communication of objects among a dynamic set of heterogeneous clients.

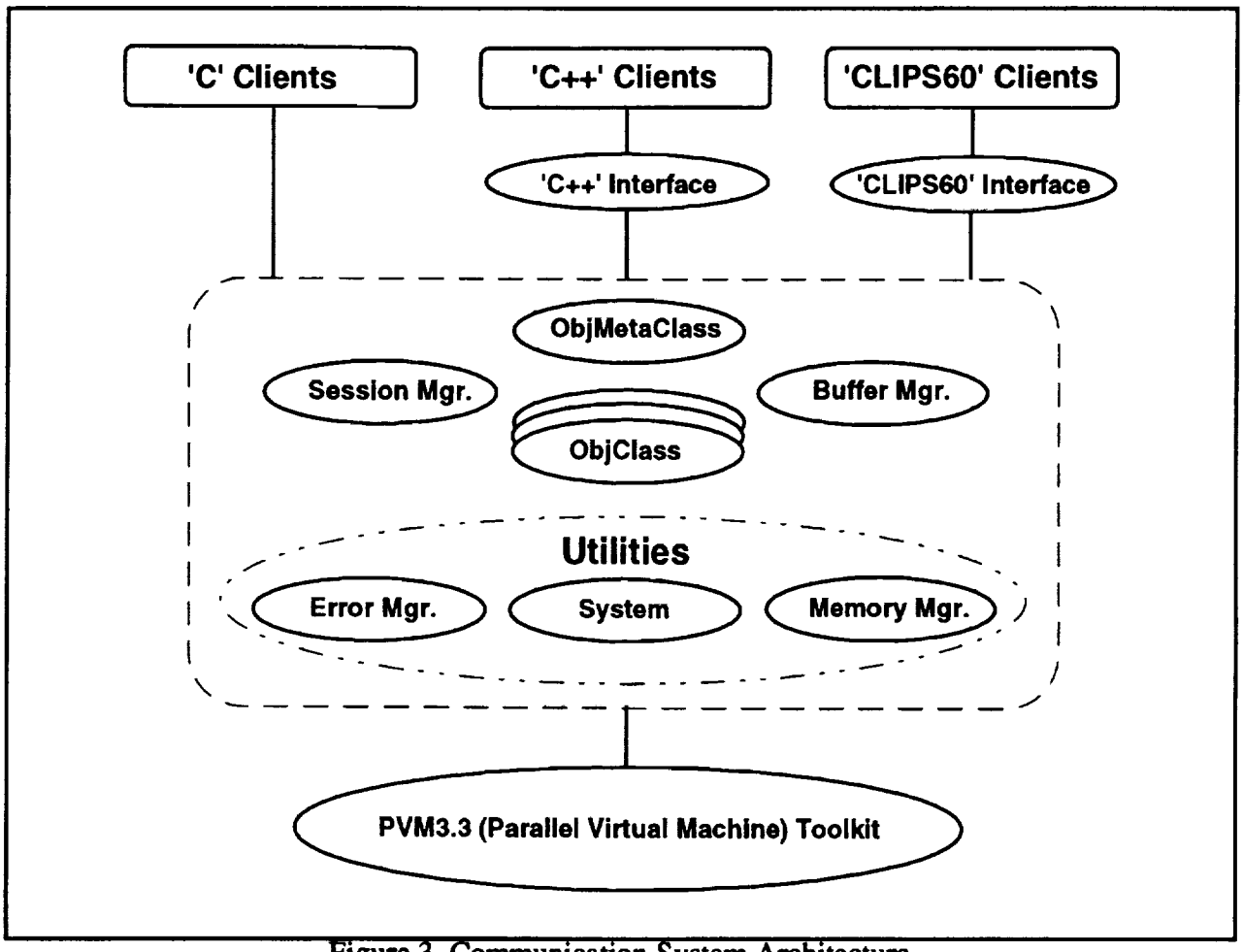


Figure 3. Communication System Architecture

Session Manager

The Session Manager (SM) allows clients to enter and exit application sessions. Within a specific session, clients can enter and exit any number of client groups. This functionality is particularly useful in sending messages to all members of a group. The SM can also invoke PVM to spawn additional application components utilizing a variety of configuration schemes. The facilities supported range from the specification of CPUs to host the spawned applications to an automatic selection of CPUs to use as hosts and balance the load across the virtual machine.

Buffer Manager

The buffer manager provides for the creation and manipulation of any number of fact buffers. Dynamic in nature, these buffers can be used to communicate several fact objects as a single atomic unit. That is, receiving CLIPS clients will accept and process the buffer's contents within

a single execution cycle. Clients are free to 'put' or 'get' facts into and out of buffers throughout the life of the buffer. Information as to the contents of a buffer can be obtained by querying the buffer directly.

Communication Object Class Library

The Communication Object Class Library contains methods which are specific to a particular data object. (These are the routines referred to in the Overview of CMS section above.) These methods include a set of constructors and destructors along with methods to 'pack' and 'unpack' an object component-by-component. This library includes additional methods to translate an object to and from the CLIPS environment and the C Object representation used with PVM. It is this class library which may require additions if new template forms are to be communicated.

Communication Object META Class

The motivation of the Communication Object META Class is twofold. The META Class combines common object class functionality into a single collection of META Class methods. That is, the highest level communication functions are exist as META Class methods, which freely accept any object defined in the Object Class Library without concern for its type. The same high-level META methods are invoked regardless of the type of objects in a communication buffer. They determine the actual type of object to be processed and then call the appropriate class method. This effectively allows clients to deal with a single set of methods independent of object type.

CLIPS Interface

Essentially acting as a client to the PVM system, this collection of modules extends the functional interface of the communication system described above to CLIPS processes. This interface provides CLIPS clients with the same functionality as their C and C++ counterparts. In an effort to preserve the syntactical style of the CLIPS assert command, several parsers were incorporated. Since the standard CLIPS external user functions interface does not support such functionality, some additional source code to the CLIPS distribution code was required. The following section provides a brief description of the functional interface presented to CLIPS clients.

CMS FUNCTIONS

ID_ObjRef **SMConnect()**
Connects the caller to a multi-agent session.

ARGUMENTS :STRING Name
LONG InitIdServerValue
LONG IdServerRange
RETURNS : A reference to the object representation of the caller or EM_ERROR_REF. Note, this object can be passed to other clients as a way to address the caller directly.

int **SMDisconnect()**
Disconnects the caller from a multi-agent session.
ARGUMENTS : void

RETURNS :EM_SUCCESS or EM_ERROR

int **SMEnterGrp()**
Enrolls the caller into the group <Group>.

ARGUMENTS : STRING Group
RETURNS : The caller's instance number within <Group> or EM_ERROR. Instance numbers start at 0 and increment upward.

int **SMExitGrp()**
Unenrolls the caller from the group, <Group>.
ARGUMENTS : void
RETURNS : EM_SUCCESS or EM_ERROR

int **SMSpawn()**
Spawns <NumCopies> copies of <Task> in accordance with the values of <How> and <where>.

ARGUMENTS : SYMBOL Task
INTEGER NumCopies
INTEGER How
<SM_DEFAULT - Target host is determined by the Session Manager. In this case <Where> should be NULL.
SM_HOST - Target host is determined by <Where>.
SM_ARCH - Target host is determined by the Session Manager. The target host's architecture will be of type <Where>.
SM_DEBUG - Target host is determined by the Session Manager. In this case <Where> should be NULL. All copies of <Task> will be run in the PVM debugger.
SM_TRACE - Target host is determined by the Session Manager. In this case <Where> should be NULL. All copies of <Task> will generate PVM trace data. >

SYMBOL Where
<Examples: "moby.cadrc.calpoly.edu" , or SUN4>

MORE ARGS ArgV
<Any # of args passed to each task upon startup.>
RETURNS : Actual number of spawned tasks or EM_ERROR.

int **BMPutTmplt()**
Places <Tmplt> into <BufRef>.

ARGUMENTS : BM_BufRef BufRef
DEFTEMPLATE Tmplt
RETURNS : EM_SUCCESS or EM_ERROR

int **BMGrpSend()**
Sends the contents of <BufRef> to each member of <Group>.
ARGUMENTS : STRING Group

BM_BufRef BufRef

RETURNS : EM_SUCCESS or EM_ERROR

int **BMQueryReceive()**
Formally processes incoming object groups. If no pending object groups exist, control is IMMEDIATELY returned to the caller.
ARGUMENTS : void
RETURNS : EM_SUCCESS or EM_ERROR

int **SGrpSendIdentity()**
Sends an identity to each member of a group.
ARGUMENTS : STRING Group
ID_ObjRef ObjRef
RETURNS : EM_SUCCESS or EM_ERROR

int **SGrpSendTmplt()**
Sends a template to each member of a group.
ARGUMENTS : STRING Group
Expression Tmplt
RETURNS : EM_SUCCESS or EM_ERROR

int **SPrintIdentity()**
Prints the object referenced by NIdentityRef> to stdout.

ARGUMENTS : ID_ObjRef IdentityRef
RETURNS : EM_SUCCESS or EM_ERROR

BM_BufRef **BMCreate()**
Creates a "send" buffer which encodes its contents according to <Encoding>.
ARGUMENTS :INTEGER Encoding
<BM_DEFAULT - Data for heterogeneous networks
BM_RAW - No encoding takes place
BM_IN_PLACE - Same as BM_DEFAULT except data is not copied into the buffer until it is sent. >
RETURNS : A reference to buffer or EM_ERROR_REF

int **BMDestroy()**
Destroys the buffer referenced by <BufRef>.

ARGUMENTS : BM_BufRef BufRef
RETURNS : EM_SUCCESS or EM_ERROR

int **BMSend()**
Sends contents of <BufRef> to <IdentityRef>. The contents of <BufRef> is left unaltered.

ARGUMENTS :ID_ObjRef IdentityRef
BM_BufRef BufRef
RETURNS : EM_SUCCESS or EM_ERROR

int **BMReceive()**
Formally processes incoming object groups. If no pending object groups exist, caller is put to sleep until such an event occurs. Received objects will be placed into the caller's fact-list in their appropriate form.

ARGUMENTS : void
RETURNS : EM_SUCCESS or EM_ERROR

int **SSendIdentity()**
Sends an identity to another client.
ARGUMENTS :ID_ObjRef DstIdentityRef
ID_ObjRef ObjRef
RETURNS : EM_SUCCESS or EM_ERROR

int **SSendTmplt()**
Sends a template to another client.
ARGUMENTS :ID_ObjRef DstIdentityRef
Expression Tmplt
RETURNS : EM_SUCCESS or EM_ERROR

Figure 4. CMS Functions

A CMS EXAMPLE

The following example illustrates a CLIPS knowledge base intended to communicate **VALUE_FRAME** templates with CMS. The example consists of four CLIPS constructs: one deftemplate and three defrules. The first step of the example is to define a **VALUE_FRAME** template having four slots. The client registers itself in the session via the **CONNECT_ME** rule. This rule also enters the client into a group. The rule then broadcasts the client's identity to all other members of its group. Incoming identifications are processed by the **RECEIVE_IDENTITY** rule. After receiving another client's identification, several **VALUE_FRAME** facts are placed into a buffer and sent back to the client.

Incoming **VALUE_FRAME** facts are processed by the **PROCESS_VALUE_FRAME** rule. Finally, when there are no more rules on the agenda, the client goes into a blocking receive state via the execution of the **RECEIVE** rule. CLIPS clients receive facts in two distinct manners. In the first case, the communication system is queried at the end of each execution cycle for pending messages. The second method by which a client can receive messages is through an explicit call to BMReceive. The functionality of this call is identical to the implicit method, with the exception that the caller will be put to sleep until a pending message exists. In either case, incoming facts are processed transparently to the client and produce immediate modifications of the fact-list.

```
(deftemplate VALUE_FRAME
    ( slot    Frame    )
    ( slot    Instance )
    ( slot    Slot     )
    ( slot    Value    )

)


(defrule RECEIVE_IDENTITY
    ( BM_DEFAULT ?Mode )
    (    SESSION-MEMBER ?Name
         CAN-BE-REFERENCED-BY ?Identity
    )
=>
    ; Send our new friend some templates
    ( bind ?BufRef ( BMCreate ?Mode ) )

    ( BMPutTmplt ?BufRef ( VALUE_FRAME
                           ( Frame  Frame1 )
                           ( Instance    Instance1 )
                           ( Slot    Slot1 )
                           ( Value  1234 )
                         )
    )
    ( BMPutTmplt ?BufRef ( VALUE_FRAME
                           ( Frame  Frame2 )
                           ( Instance    Instance2 )
                           ( Slot    Slot2 )
                           ( Value  56.789 )
                         )
    )
    ( BMPutTmplt ?BufRef ( VALUE_FRAME
                           ( Frame  Frame3 )
                           ( Instance    Instance3 )
                           ( Slot    Slot3 )
```

```
(defrule CONNECT_ME
    ( initial-fact )
=>
    ( bind ?MyName "Access" )
    ( bind ?MyGrp "AgentGrp" )

    ; Connect to the session
    ( bind ?MyId ( SMConnect ?MyName 200 200) )

    ; Join a group
    ( SMEnterGrp ?MyGrp )

    ; Send my identity to members of my group
    ( SGrpSendIdentity ?MyGrp ?MyId)


    ; Assert the "receive" control fact

    ( assert( RECEIVE ) )
)


(defrule PROCESS_VALUE_FRAMES
    ?TMPLT <- ( VALUE_FRAME
                ( Frame ?Frame )
                ( Instance ?Instance )
                ( Slot ?Slot )
                ( Value ?Value )
              )
=>
    ; Process the template

        ...

    ( retract ?TMPLT )
)
```

```
                    ( Value  "How are you ?" )
                )                                    (defrule RECEIVE
    )                                                    (declare (salience -10000 ) )
    ; Send the buffer to our new friend?                 ?REC <- ( RECEIVE )
    ( BMSend ?Identity ?BufRef )                     =>
                                                         ( BMReceive )
    ; Destroy the buffer                                 ( retract ?REC )
    ( BMDestroy ?BufRef )                                ( assert( RECEIVE ) )
)                                                    )
```

Figure 5. A CMS Sample

CONCLUSION

PVM and CLIPS both provide free source code systems that are well maintained by developers and a sizable number of users. Relative few source code changes are necessary to either system in order to build a reliable and robust platform that will support distributed computing in a heterogeneous environment of CPUs operating under UNIX. The CMS system described in this paper provides the CLIPS interface code and some parsing code sufficient to enable efficient use of PVM facilities and communication of CLIPS facts and templates among C, C++, and CLIPS processes within a PVM virtual machine. Even more efficient communication can be obtained through enhancements to the PVM source code that can provide more efficient allocation of memory and reuse of PVM message buffers in certain applications.

NOTES

Information on PVM is best obtained by anonymous ftp from: netlib2.cs.utk.edu
Shar and tar packages are available from the same source.

The authors are currently using the CMS system in applications that involve multiple CLIPS expert systems in sophisticated interactive user interface settings. It is expected that the basic CMS code will become available in the Spring, 1995. Inquiries via e-mail are preferred.

BIBLIOGRAPHY

1. Riley, Gary, B. Donnell et. al., "CLIPS Reference Manual," JSC-25012, Lyndon B. Johnson Space Center, Houston, Texas, June, 1993.

2. Pohl, Jens, A. Chapman, L.Chirica, R. Howell, and L. Myers, "Implementation Strategies for a Prototype ICADS Working Model," CADRU-02-88, CAD Research Unit, Design Institute, School of Architecture and Design, Cal Poly, San Luis Obispo, California, June, 1988.

3. Myers, Leonard and J. Pohl, "ICADS Expert Design Advisor: An Aid to Reflective Thinking," Knowledge-Based Systems, London, Vol. 5, No. 1, March, 1992, pp. 41-54.

4. Pohl, Jens and L. Myers, "A Distributed Cooperative Model for Architectural Design," Automation In Construction, Amsterdam, 3, 1994, pp. 177-185.

6. Geist, Al, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam, "PVM 3 User's Guide and Reference Manual," ORNL/TM-12187, Oak Ridge National Laboratory, Oak Ridge . Tennessee, May, 1993.

A Parallel Strategy for Implementing Real-Time Expert Systems Using CLIPS

Laszlo A. Ilyes & F. Eugenio Villaseca
Cleveland State University
Dept. of Electrical Engineering
1983 East 24th Street
Cleveland, Ohio 44115

John DeLaat
NASA Lewis Research Center
21000 Brookpark Road
Cleveland, Ohio 44135

ABSTRACT

As evidenced by current literature, there appears to be a continued interest in the study of real-time expert systems. It is generally recognized that speed of execution is only one consideration when designing an effective real-time expert system. Some other features one must consider are the expert system's ability to perform temporal reasoning, handle interrupts, prioritize data, contend with data uncertainty, and perform context focusing as dictated by the incoming data to the expert system.

This paper presents a strategy for implementing a real time expert system on the iPSC/860 hypercube parallel computer using CLIPS. The strategy takes into consideration, not only the execution time of the software, but also those features which define a true real-time expert system. The methodology is then demonstrated using a practical implementation of an expert system which performs diagnostics on the Space Shuttle Main Engine (SSME).

This particular implementation uses an eight node hypercube to process ten sensor measurements in order to simultaneously diagnose five different failure modes within the SSME. The main program is written in ANSI C and embeds CLIPS to better facilitate and debug the rule based expert system.

INTRODUCTION

Strictly defined, an expert system is a computer program which imitates the functions of a human expert in a particular field [1]. An expert system may be described as a real-time expert system if it can respond to user inputs within some reasonable span of time during which input data remains valid. A vast body of recently published research clearly indicates an active interest in the area of real-time expert systems [2-12].

Science and engineering objectives for future NASA missions require an increased level of autonomy for both onboard and ground based systems due to the extraordinary quantities of information to be processed as well as the long transmission delays inherent to space missions. [13]. An expert system for REusable Rocket Engine Diagnostics Systems (REREDS) has been investigated by NASA Lewis Research Center [14, 15, 16]. Sequential implementations of the expert system have been found to be too slow to analyze data for practical implementation. As implemented sequentially, REREDS already exhibits a certain degree of inherent parallelism. Ten

USING CLIPS IN THE DOMAIN OF KNOWLEDGE-BASED MASSIVELY PARALLEL PROGRAMMING

Jiri J. Dvorak
Section of Research and Development
Swiss Scientific Computing Center CSCS
Via Cantonale, CH-6928 Manno, Switzerland
Email: dvorak@cscs.ch

Abstract

The *Program Development Environment* PDE is a tool for massively parallel programming of distributed-memory architectures. Adopting a knowledge-based approach, the PDE eliminates the complexity introduced by parallel hardware with distributed memory and offers complete transparency in respect of parallelism exploitation. The knowledge-based part of the PDE is realized in CLIPS. Its principal task is to find an efficient parallel realization of the application specified by the user in a comfortable, abstract, domain-oriented formalism. A large collection of fine-grain parallel algorithmic skeletons, represented as COOL objects in a tree hierarchy, contains the algorithmic knowledge. A hybrid knowledge base with rule modules and procedural parts, encoding expertise about application domain, parallel programming, software engineering, and parallel hardware, enables a high degree of automation in the software development process.
In this paper, important aspects of the implementation of the PDE using CLIPS and COOL are shown, including the embedding of CLIPS with C++-based parts of the PDE. The appropriateness of the chosen approach and of the CLIPS language for knowledge-based software engineering are discussed.

1 INTRODUCTION

Massive parallelism is expected to provide the next substantial increase in computing power needed for current and future scientific applications. Whereas there exists a variety of hardware platforms offering massive parallelism, a comfortable programming environment making programming of distributed-memory architectures as easy as programming single address space systems is still missing. The programmer of parallel hardware is typically confronted with aspects not found on conventional architectures, such as data decomposition, data and load distribution, data communication, process coordination, varying data access delays, and processor topology. Approaches to a simplification of parallel programming that have been used previously, in particular for shared-memory architectures, are transformational and compile-time parallelization [Pol88, ZC90]. However, such code-level parallelization is extremely difficult for distributed architectures. Additionally, only a limited, fine-grain part of the opportunities for parallel execution within the program can be detected, and the results are dependent on the programming style of the programmer. The detection of conceptual, coarse-grain parallelism found in

12. Hota, K., Nomura, H., Takemoto, H., Suzuki, K., Nakamura, S., and Fukui, S., "Implementation of a Real-Time Expert System for a Restoration Guide in a Dispatching Center," IEEE TRANSACTIONS ON POWER SYSTEMS, Vol. 5, No. 3, 1990, pp. 1032-8.

13. Lau, S. and Yan, J. , "Parallel Processing and Expert Systems," NASA Technical Memorandum, No. 103886, May, 1991.

14. Merrill, Walter and Lorenzo, Carl, "A Reusable Rocket Engine Intelligent Control," 24th Joint Propulsion Conference, Cosponsored by AIAA, ASME, and ASEE, Boston MA, July 1988.

15. Guo T.-H. and Merrill, W., "A Framework for Real-Time Engine Diagnostics," 1990 Conference for Advanced Earth-to-Orbit Propulsion Technology," Marshall Space Flight Center, May 1990.

16. Guo, T.-H., Merrill, W. , and Duyar, A., "Real-Time Diagnostics for a Reusable Rocket Engine," NASA Technical Memorandum No. 105792, August 1992.

17. Kreidler, D. and Vickers, D., "Distributed Systems Status and Control," Final Report to NASA Johnson Space Center, NASA CR 187014, September 1990.

18. Anex, Robert, "Reusable Rocket Engine Diagnostic System Design,"Final Report, NASA CR 191146, January 1993.

19. Randall, M.R., Barkadourian, S., Collins, J.J., and Martinez, C., "Condition Monitoring Instrumentation for Space Vehicle Propulsion Systems," NASA CP 3012, pp. 562-569, May 1988.

20. CLIPS 6.0 User's Manual, Volume II, Advanced Programming Guide, pp. 43-142, June, 1993.

Research Center, as well as Mr. Gary Riley of NASA Johnson Space Center. It is only with their patient and supportive contributions that this research was possible.

REFERENCES

1. Lugger, George and Stubblefield, William, "AI: History and Applications," ARTIFICIAL INTELLIGENCE AND THE DESIGN OF EXPERT SYSTEMS, The Benjamin/Cummings Publishing Co. Inc., New York, NY, 1989, pp. 16-19.

2. Leitch, R., "A Real-Time Knowledge Based System for Process Control," IEE PROCEEDINGS, PART D: CONTROL, THEORY AND APPLICATIONS, Vol. 138, No. 3, May, 1991, pp. 217-227.

3. Koyama, K.,"Real-Time Expert System for Gas Plant Operation Support (GAPOS)," 16TH ANNUAL CONFERENCE OF IEEE INDUSTRIAL ELECTRONICS, November, 1990.

4. Bahr, E., Barachini, F., Dopplebauer, J. Grabner, H. Kasperic, F., Mandl, T., and Mistleberger, H., "Execution of Real-Time Expert Systems on a Multicomputer," 16th ANNUAL CONFERENCE OF IEEE INDUSTRIAL ELECTRONICS, November, 1990.

5. Wang, C. Mok, A., and Cheng, A., "A Real-Time Rule Based Production System," PROCEEDINGS OF THE 11TH REAL-TIME SYSTEMS SYMPOSIUM, December, 1990.

6. Moore, R., Rosenhof, H., and Stanley, G., "Process Control Using Real-Time Expert System," PROCEEDINGS OF THE 11TH TRIENNIAL WORLD CONGRESS OF THE INTERNATIONAL FEDERATION OF AUTOMATIC CONTROL, August, 1990.

7. Lee, Y., Zhang, L., and Cameron, R., "ESC: An Expert Switching Control System," INTERNATIONAL CONFERENCE ON CONTROL, March, 1991.

8. Jones, A., Porter, B., Fripp, R., and Pallet, S., "Real-Time Expert System for Diagnostics and Closed Loop Control," PROCEEDINGS OF THE 5TH ANNUAL IEEE INTERNATIONAL SYMPOSIUM ON INTELLIGENT CONTROL, September, 1990.

9. Borsje, H., Finn, G., and Christian, J., "Real-Time Expert Systems and Data Reconciliation for Process Applications," PROCEEDINGS OF THE ISA 1990 INTERNATIONAL CONFERENCE AND EXHIBITION, Part 4 of 4, October, 1990.

10. Silagi, R. and Friedman, P., "Telemetry Ground Station Data Servers for Real-Time Expert Systems," INTERNATIONAL TELEMETERING CONFERENCE, ITC/USA, October, 1990.

11. Spinrad, M., "Facilities for Closed-Loop Control in Real-Time Expert Systems for Industrial Automation," PROCEEDINGS OF THE ISA/1989 INTERNATIONAL CONFERENCE AND EXHIBIT, Part 2 of 4, October, 1990.

(nearly) simultaneous data requests. By adding a second server node to the system, this contention can be greatly reduced.

Since the data can be processed at a fast, continuous rate, the validity of sensor measurements can be assured during processing. Consequently, truth maintenance is realized by suppressing data requests to the server node until all sensor measurements have been simultaneously updated. This guarantees that all data accessed by the failure detector nodes during any processing cycle is the same "age."

Due to the nature of the particular expert system selected for this research, the time required by the failure detectors to process SSME data remains constant regardless of whether or not a failure condition exists. Thus, predictability is always assured for this example. Also, the need for temporal reasoning is not explicitly indicated and is therefore not investigated. Since these aspects of the design are application specific, they and must be investigated in future work using different expert system models.

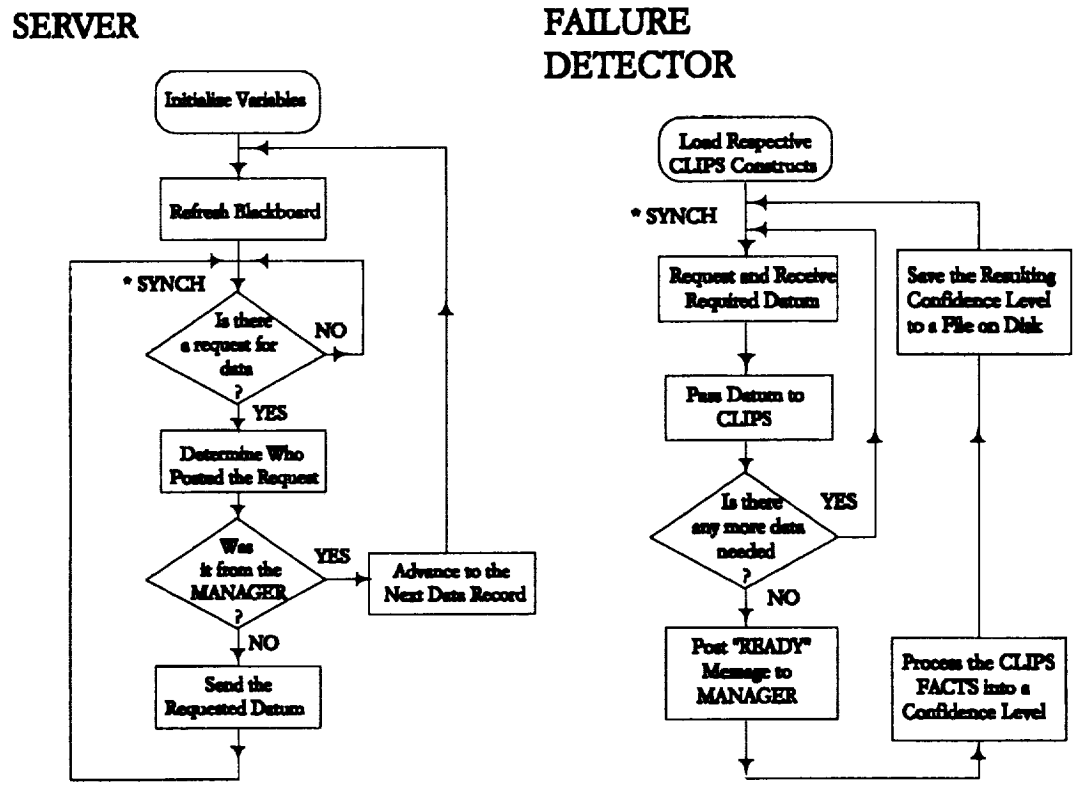
As discussed earlier, uncertainty handling is inherent to this expert system. The voting scheme and use of confidence levels permits reasoning, even in the presence of noisy, incomplete, or inconsistent data. Since the output from the system is a graded value rather than a binary value, the output carries with it additional information about the expert system's confidence that a particular failure is occurring.

One of the most important features of this design is that program flow control and system I/O is accomplished in C language code. Using CLIPS as an embedded application within a fast, compiled body of C language code allows the expert system to be more easily integrated into a practical production system. Complex reasoning can be relegated directly and exclusively to the nodes invoking the CLIPS environment, while tasks which are better suited to C language code can be performed by the server and manager nodes. Thus, simple decisions can be realized quickly in C language rather than relying on the slower CLIPS environment. Based on fast preprocessing of the sensor measurements, the C language code can be used to initiate process interrupts during emergency conditions and even change the context focusing of the expert system. Those tasks which require complex reasoning can be developed and refined separately in CLIPS, taking full advantage of the debugging tools available in the CLIPS development environment.

While the rules for this particular expert system are somewhat simple compared to other applications considered in the literature, it is believed that the approach used in this study can be extended to other examples. This study demonstrates that parallel processing can not only speed up the execution of certain expert systems, but also incorporate other important features essential for real-time operation.

ACKNOWLEDGMENTS

We would like to thank NASA Lewis Research Center for its financial support of this research under cooperative agreement NCC-257 (supplement no. 3). We would also like to extend our thanks to the technical support staff of the ACCL and the internal reviewers at NASA Lewis



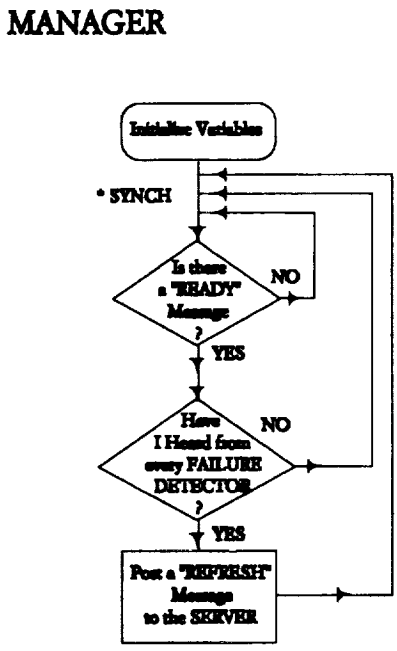


Figure 2. - Process flow for the a) Server Node, b) Failure Detector Nodes, and c) Manager Node

Each individual failure detector was implemented in CLIPS on a personal computer and its accuracy was tested and verified using simulated SSME sensor data. Once satisfactory results were achieved, an ANSI C program was written for the iPSC/860 hypercube computer which would initialize the CLIPS environment on five nodes of a 2^3 hypercube structure. These five nodes, referred to as the *failure detector nodes*, load the constructs for one failure detector each, and use CLIPS as an embedded application as described in the CLIPS Advanced Programming Guide [20]. In this way, CLIPS will only be used for evaluation of the REREDS rules. All other programming requirements, including opening and closing of sensor measurement data files, preliminary data analysis, and program flow control are handled in C language. By embedding the CLIPS modules within ANSI C code, context focusing and process interruptions can be more efficiently realized.

Coordination of data acquisition and distribution among the failure detector nodes is accomplished through a *server node* which is programmed to furnish sensor measurement data to requesting nodes. Since the data for this study originate from the SSME simulator test bed, data retrieval is accomplished simply by reading sequential data from prepared data files. The server node transfers incoming sensor measurements into an indexed memory array, or blackboard, from which data are distributed upon request to the failure detector nodes. When the blackboard is updated, all requests for data are ignored until data transfer is completed. This assures that reasoning within the expert system is always performed on contemporaneous data. The server node does not invoke the CLIPS environment at any time. It is programmed entirely in C language code.

One additional node, referred to as the *manager node*, is used by the expert system to coordinate the timing between the failure detector nodes and the server node. Like the server node, the manager node does not invoke the CLIPS environment. Once the manager node has received a "ready" message from all failure detector nodes, it orders the server node to refresh the data in the blackboard. During this refresh, the failure detector nodes save their results to permanent storage on the system. The activities and process flow of all three types of nodes used in this research are illustrated in Figure 2. The asterisk denotes the point at which all nodes synchronize.

CONCLUSION

Profiling studies were conducted on the parallel implementation of the REREDS expert system. It was found that the system could process the sensor measurements and report confidence levels for all five failure modes in 18 milliseconds. A sequential implementation of the expert system on the same hardware was found to require over 50 milliseconds to process and report the same information, indicating that the parallel implementation can process data at nearly three times as quickly. Considering the fact that seven processors are being used in the parallel implementation, these results may seem somewhat disappointing, however, the profiling studies also indicate that additional speedup can be realized in future implementations of this expert system if the data blackboard is also parallelized. Using only one server node causes some hardware contention. Shortly after the nodes synchronize, the failure detectors tend to overwhelm the server with five

Once a vote has been assigned to every sensor measurement, each failure detector averages the votes for all of its corresponding sensor measurements. The final result will be a number between -1.00 and +1.00. This result is converted to a percent and is

Failure Detector	Description	Measurements	Failure States
F11/15	Labyrinth/Turbine Seal Leak	LPFP Discharge Pressure	Low
		FPOV Valve Position	High
		HPFTP Turbine Discharge Temp.	High
F67	HPOTP Turbine Interstage & Tip Seal Wear	HPOTP Discharge Temperature	Low
		HPOP Discharge Pressure	Low
		HPOTP Shaft Speed	Low
		MCC Pressure	Low
F68	Intermediate Seal Wear	Secondary Seal Drain Temperature	Low
		HPOTP Inter-Seal Drain Pressure	Low
F69	HPOP Primary Seal Wear	HPOP Primary Seal Drain Pressure	High
		Secondary Seal Drain Pressure	High
		Secondary Seal Drain Temperature	High
F70	Pump Cavitation	HPOP Discharge Pressure	Low
		HPOTP Shaft Speed	Low
		MCC Pressure	Low

Table 1. - Failure Detectors Only Requiring Sensor Measurements for Failure Diagnosis

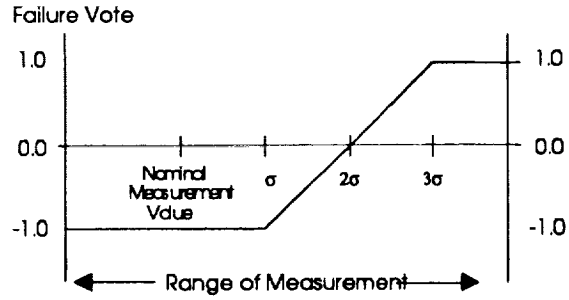


Figure 1. - Voting Curve for Sensor Measurement with "High" Failure State

referred to as the corresponding confidence level of that failure mode. The underlying motivation for this approach is to add inherent uncertainty handling to the expert system.

sensor measurements are used to diagnose the presence of five different failures which may manifest themselves in the working SSME. Each module of code which diagnoses one failure is referred to as a failure detector. While some of the sensor measurements are shared between failure detectors, the computations within these detectors are completely independent of one other.

One apparent way to partition the problem of detecting failures in the SSME, is to assign each failure detector to its own node on the hypercube system. Because the failure detectors may be processed simultaneously, a speedup in the execution is expected. But while execution time is a critical parameter in any real-time expert system, it is not the only ingredient required in order to guarantee its success. A recent report characterized the features required of expert systems to operate in real-time. In addition to the requirement of fast execution, the real-time expert system should also possess the ability to perform context focusing, interrupt handling, temporal reasoning, uncertainty handling, and truth maintenance. Furthermore, the computational time required by the system should be predictable and the expert system should potentially be able to communicate with other expert systems [17]. These aspects are considered in the design presented in this paper.

METHODS

The rules for diagnosing failures in the SSME were elicited from NASA engineers and translated into an off-line implementation of a REREDS expert system [18]. While some of the failures can be diagnosed using only sensor measurements, other failures require both data measurements and the results obtained from condition monitors. The condition monitors measure both angular velocity and acceleration on various bearings of the High Pressure Oxidizer Turbo-Pump (HPOTP) shaft and determine the magnitudes of various torsional modes in the HPOTP shaft [19]. Due to the lack of availability of high frequency bearing data and additional hardware requirements for implementing real-time condition monitors, this expert system considered only those failure detectors which required sensor measurements alone.

The five failure detectors which rely solely on sensor measurements for diagnosis are listed in Table 1 along with a description of the failure, the required sensor measurements, and their respective, relative failure states. Notice that failure detectors designated F11 and F15 cannot be differentiated from one another and are thus combined into one single failure mode.

For each sensor measurement listed, the expert system knowledge base is programmed with a set of nominal values and deviation values (designated in our work by σ). One of the roles of the expert system is to match incoming sensor measurements with the nominal and deviation values which correspond to the specific power level of the SSME at any given time. Any sensor measurement may deviate from the nominal value by $\pm \sigma$ without being considered high or low relative to nominal. Beyond the σ deviation, the sensor measurement is rated with a value which is linearly dependent upon the amount of deviation. This value is referred to as a vote and is used by a failure detector to determine a confidence level that the failure mode is present. This voting curve is illustrated in Figure 1.

many applications from natural sciences and engineering is in general too complicated for current parallelization techniques.

Our *Program Development Environment* PDE [Dvo93, DDR94] is a tool for programming massively parallel computer systems with distributed memory. Its primary goal is to handle all complexity introduced by the parallel hardware in a user-transparent way. To break the current limitations in code parallelization, we approach parallel program development with knowledge-based techniques and with user support starting at an earlier phase in program development, at the specification and design phase.

In this paper we focus on the expert system part of the PDE realized in the CLIPS [GR94] language. At first, a short overview of the PDE is given. The representation and use of knowledge is the topic of section 3. This is followed by sections showing aspects of external interfaces, problem representation, and embedding with C++. The important role of an object-oriented approach will be elaborated in these parts. A discussion of results, the appropriateness of the CLIPS language, and the current state of the development concludes the paper.

2 OVERVIEW OF THE PDE

The basic methodology of programming with the PDE consists of three steps [DR92]:

1. Problem specification using a domain-specific, high-level formalism
2. Interactive refinement and completion of the specification (if needed)
3. User-transparent generation of compilable program code

According to this three-step approach to the programming process, the program development environment consists of the three functional components shown in Fig. 1. In a typical

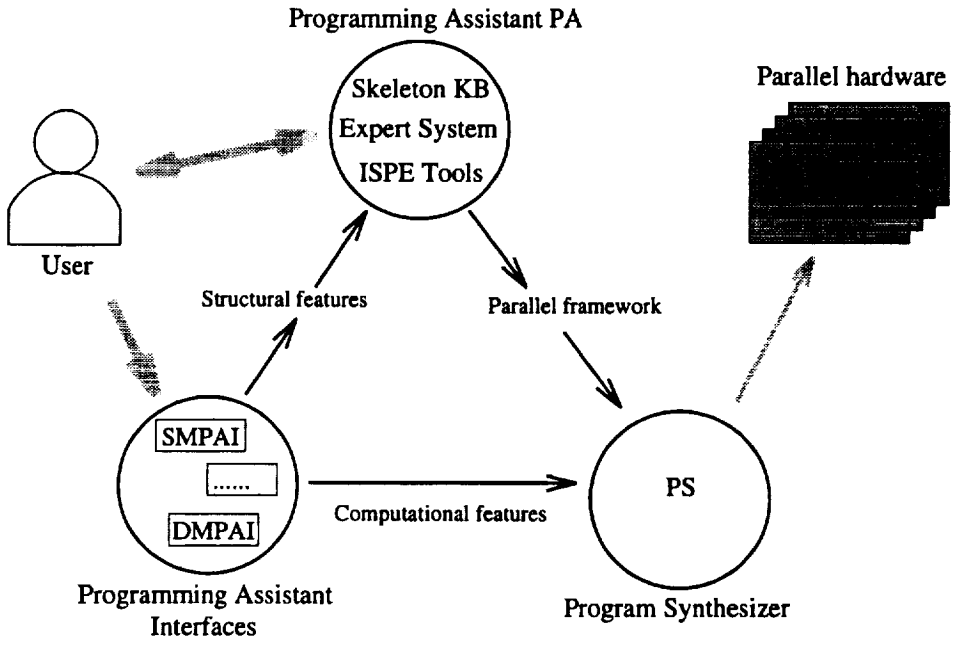


Figure 1: The conceptual structure of the PDE

session, the programmer gives an initial specification of the problem under consideration using a *programming assistant interface.* Currently, we have an interface *SMPAI* covering the domain of stencil-based applications on n-dimensional grids. Other interfaces are in preparation. The initial problem specification is decomposed into the purely computational features and the features relevant for the parallel structure. The first are passed directly to the *Program Synthesizer PS*, the latter go to the *Programming Assistant PA.* Then, in interaction with the user, the PA extracts and completes the information needed to determine an appropriate parallel framework. The PA is the central, largely AI-based component of the PDE, relying on various kinds of expert knowledge. The program synthesizer expands the parallel framework into compilable, hardware specific, parallel C++ or C programs.

3 REPRESENTATION OF ALGORITHMIC KNOWLEDGE

The PDE embodies a skeleton-oriented approach [Col89] to knowledge-based software engineering. A large collection of hierarchically organized fine-grain skeletons forms the algorithmic knowledge. The hierarchy spans from a general, abstract skeleton at the root to highly problem-specific, optimized skeletons at the leaf-level. Skeleton nodes in the tree represent a number of different entities. First, every non-leaf skeleton is a decision point for the rule-based descent. The descendants are specializations of the actual node in one particular aspect, the *discrimination criterion* of the skeleton. Every skeleton node contains information about the discrimination criterion and value that leads to its selection among the descendants of its parent node. The discrimination criteria represent concepts from parallel programming, application domain, and characteristics the supported parallel hardware architectures. Second, a skeleton node holds information about requirements other than the discrimination criterion that have to be checked before a descent is done. Third, skeleton nodes have slots for other kind of information that is not checked during descent but that may be helpful for the user or for subsequent steps. Finally, the skeleton nodes have attached methods or message handlers that are able to dynamically generate the parallel framework for the problem to be solved with the chosen skeleton.

Based on these different roles, the question arises whether the skeleton knowledge base should be built using instance or class objects. To access information in slots, a skeleton node should be an instance of a suitable class. However, to attach specialized methods or message-handlers at any node, skeletons have to be represented using classes. Also, the specialization type of the hierarchy suggests the use of a class hierarchy for the skeleton tree. We have solved the conflict by using both a class and an instance as the representation of a single skeleton node. A simplified example skeleton class is:

```
(defclass skel-32 (is-a init-bnd-mixin skel-5)
  (pattern-match reactive)
  (slot discrimination-crit (default problem-type) (create-accessor read))
  (slot discrimination-value (default INIT_BND_VAL_PROBLEM)
                             (create-accessor read))
  (multislot constraints (create-accessor read-write))
  (multislot required (default (create$ gridinstances_requirement))
                      (create-accessor read)))
```

The `required` slot holds instances of the requirements mentioned above. Constraints are

an additional concept used for describing restrictions in application scope.

All instances of the skeleton nodes are generated automatically with a simple function:

```
(deffunction create-instances (?rootclass)
  (progn$ (?node (class-subclasses ?rootclass inherit))
    (make-instance ?node of ?node)))
```

The two basic operations on the skeleton tree are the finding of the optimal skeleton, i.e., the descent in the tree, and the generation of the computational framework based on a selected node.

3.1 Rule-based skeleton tree descent

The descent in the skeleton tree is driven by rules. Based on the discrimination criterion of the current node, rules match on particular characteristics of the application specification and try to derive a value for the criterion. If a descendant node with the derived criterion value exists, it will get the candidate for the descent. The following example rule tries to find a descendant matching the grid coloring property of the specification:

```
(defrule DESC::coloring-1
  ?obj <- (object (name [descent_object])
                  (curr_skel ?skel))
  ?inst <- (object (discrimination-crit coloring)
                   (discrimination-value ?c))
  (test (member$ (class ?inst) (class-subclasses (class ?skel))))
  ?res <- (object (nr_of_colors ?c))
  ?app <- (object (is-a PAapp_class)
                  (results ?res))
  =>
  (send ?obj put-next_skel ?inst)
  (send ?obj put-decision_ok t))
```

The pattern matching relies heavily on object patterns. The whole expert system is programmed practically without the use of facts. The rule first matches a descent object, then a skeleton which is a descendant of the current skeleton and finally the `nr_of_colors` slot of the problem specification. Note that the particular ordering of object patterns is induced by some peculiarities of the CLIPS 6.0 pattern matching procedure. In order to handle references to objects in other modules, it is advisable to use instance addresses instead of instance names. CLIPS does not match objects from imported modules when they are refered to by a name property and the module name is not explicitly given. On the other hand, CLIPS has no instance-address property for object patterns and the form `?obj <- (object ...)` does not allow `?obj` to be a bound variable. So, the ordering of patterns is restricted in such a way that first an object gets matched and then it is verified whether it is the desired one. The consequence is probably some loss in pattern matching efficiency.

Although our initial problem domain is sufficiently restricted and well-defined to allow a complete, automatic descent in most cases, there is a collection of user interaction tools, called the *Intelligent Skeleton Programming Environment* ISPE, ready to handle the case

when the descent stops before reaching a sufficiently elaborated skeleton. With the ISPE, the user can add missing information, select a descendant node manually, or even step through the tree guided by the expert system. For such a functionality, the ISPE needs a high integration with the expert system. The rules for the descent are divided into various modules, separating search for descendants, verification and actual descent in different rule modules. The object of class `descent_class` used in the rule above serves to keep track of the current state of descent and to coordinate between the rule modules.

3.2 Generating output from the skeleton tree

After successful selection of a skeleton node for the application under consideration, the parallel framework can be generated. This is done by calling message-handlers attached to all skeletons. An example parallel framework for a simple stencil problem is:

```
init(Grid);
FOR (iter = 0; iter < nr_of_iterations; iter++) DO
 FOR (color = 0; color < 3; color++) DO
  INTERACTION
      fill_buf(Grid, w_obuf, west, color);
      Exchange(e_ibuf, east, w_obuf, west);
      scatter_buf(Grid, e_ibuf, east, color);
      .....
  END;
  CALCULATION
      update(Grid, color);
  END;
 ENDFOR;
ENDFOR;
finish(Grid);
```

The building blocks of this formalism [BG94] are communication and computation procedure calls, grouped by sequencing or iteration statements.

Object-oriented concepts are realized for optimizing the representational efficiency in the skeleton tree. First, *inheritance* insures that information stored in the slots of the skeleton nodes is passed down the tree. If from a certain point on it does not apply any more, it can be overriden. Second, new behaviour is introduced with *mixin classes*. As it can be seen in the `skel-32` node shown earlier, each node has both a mixin class and the parent class in its superclass list. In this way, it is possible to define a particular feature only once, but to add it at various points in the tree. Finally, instead of having methods that for each skeleton individually generate the complete parallel framework, an incremental *method combination* approach has been chosen. Every method first calls the same method of its parent node and then makes its own additions. The message-handler below shows this basic structure:

```
(defmessage-handler stencil-MS-mixin generate-MS ()
  (bind ?inst (call-next-handler))
  (send ?inst put-global_vars
        (create$ .... )))
```

Message-handlers instead of methods have to be used for this kind of incremental structure, as methods in CLIPS do not have dynamic precedence computation, whereas message-handlers do. Using methods, the sequence of skeleton and method definitions in the source file would be dominant for the precedence instead of the class hierarchy at runtime.

4 INTERFACES AND PROBLEM REPRESENTATIONS

The expert system of the PDE has basically three interfaces to the outside. It receives input from the problem specification tools, generates the parallel framework as output for the program synthesizer component, and it has an interface for user interaction.

4.1 Input

In order to omit a parser written in CLIPS, the formalism for the problem specification was chosen to be directly readable into CLIPS. A natural way to achieve this consists in using instance-generating constructs. We have two types of constructs in the input formalism, for example:

```
(make-instance global_spec of global_spec_class
               (grid_const 1.0)
               (dimension 2)
               (problem_type BND_VAL_PROBLEM))

(stencil_spec (assign_op (grid_var f (coord 0 0))
                         (function_call median
                                        (grid_var f (coord -1 0))
                                        (grid_var f (coord 0 0))
                                        (grid_var f (coord 1 0)))))
```

In the first statement above, where global properties of the application are defined, the `make-instance` is part of the formalism. Obviously, reading such a construct from a file with the CLIPS `batch` command generates an instance of the respective class and initializes the slots. In the second case, where the stencil is defined, the instance-generating ability is not directly visible. However, for every keyword such as `stencil_spec`, `assign_op` or `grid_var`, there are functions creating instances of respective classes, e.g.:

```
(deffunction grid_var (?name ?coord)
  (bind ?inst (make-instance (gensym*) of grid_var))
  (send ?inst put-var_name ?name)
  (send ?inst put-coord ?coord))
```

The reasons to not use `make-instance` in all cases are that some constructs can have multiple entities of the same name, e.g., a stencil can have multiple assignment operations, and additional processing that is needed by some constructs.

The lack of nested lists in CLIPS was considered as a disadvantage at the beginning, in particular regarding the close relation between CLIPS and Lisp/CLOS. However, a recursive list construct can be easily defined with COOL objects and instance-generation

functions similar to the one above. Certainly, this is not appropriate if runtime efficiency is critical. In the course of the PDE development, the lack of lists proved to have a positive effect on style and readability. For example, instead of using just the list `(-1 1 2)` for a 3-D coordinate, using the construct `(coord -1 1 2)` creates an instance of a specialized class for coordinate tuples. Such an instance can be easier handled than a list and appropriate methods or message-handlers can be defined. The whole problem representation after reading the specification input is present in a number of nested, interconnected instances of respective classes.

4.2 Output

The result of the generation of the parallel framework is a problem representation by means of a number of instances, much in the sense of the input representation shown above. The writing of an output file as shown in section 3.2 is done with message-handlers attached to each of the relevant problem representation classes, e.g., a for-loop is written by:

```
(defmessage-handler for_loop_class write-out (?stream)
  (printout ?stream "  FOR (" ?self:varname " = " ?self:from
                    "; " ?self:varname " < " ?self:till
                    "; " ?self:varname "++) DO " t)
  (progn$ (?el ?self:subs)
          (send ?el write-out ?stream))
  (printout ?stream "  ENDFOR;" t))
```

5 EMBEDDING WITH C++

The global structure of the PDE consists of components written in C++, among them the main program and the graphical user interface, and an embedded CLIPS expert system for knowledge representation and reasoning. Additionally, two parsers use the Lex/Yacc utilities. Whereas some components, such as the parsers, are integrated only by means of intermediate files for input and output, the expert system is highly integrated with the C++-based ISPE and the graphical user interface. The CLIPS dialog itself is visible to the user through a CLIPS base window. Figure 2 shows both the CLIPS dialog window and a browser window for graphically browsing the skeleton tree.

5.1 Data integration

Both CLIPS and C++ offer objects for the representation of data. It is therefore a straightforward decision to use the object mechanism for the data integration between an expert system written in CLIPS and programs written in C++. The concept for the CLIPS-C++ integration relies on the decisions to represent common data on the CLIPS side using COOL objects and to provide wrapper classes on the C++ side for a transparent access to COOL objects. A class hierarchy has been built in C++ to represent CLIPS types, including classes and instances. Access to COOL classes is needed for example in the skeleton tree browser, where descendants of a node are only found by inspecting the subclasses list of the node.

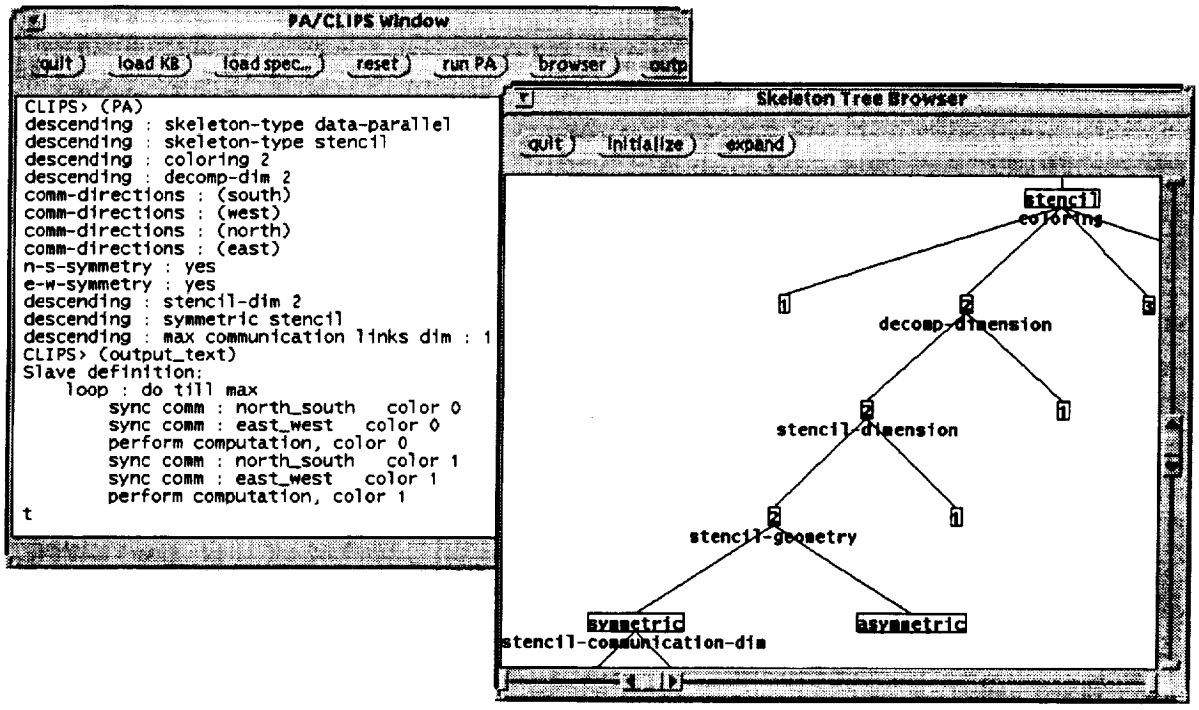


Figure 2: The CLIPS window and the skeleton tree browser

The C++ wrapper classes consist of an abstract class `clips_type` with subclasses for the CLIPS data types integer, float, symbol, string, multifield, and for COOL classes and instances. The class `coolframe` used to represent COOL instances is shown below:

```
class coolframe : public clips_type {
  char* framename;
 public:
  coolframe(char* name);
  clips_type* get_slot(char* slotname);
  int put_slot(char* slotname, clips_type* value);
  char* class_of(); };
```

The creation of a C++ frontend to a COOL instance is performed by just instantiating the above class, giving the constructor the name of the COOL instance. Accesses to slots have to be done by using the `get_slot` and `put_slot` member functions that refer to functions in the CLIPS external interface [NAS93]. The class `coolframe` is a generic class, usable for any COOL instance, no matter what collection of slots the COOL instance has. A more sophisticated approach with separate members for all slots would be somewhat more convenient, as the distinction between accessing a C++ object and accessing a COOL object through the C++ wrapper would be completely blurred. However, defining classes on the fly, based on the structure of the COOL instances, is not possible in C++. The chosen system with general-purpose wrapper classes is already convenient for use in C++. Operators and functions in C++ can be overloaded to handle CLIPS data transparently. For example, the basic mathematical operations can be overloaded to combine C++ numbers with CLIPS numbers in one expression.

Care has to be taken to not introduce inconsistencies between the data stored in CLIPS and the C++ wrappers. To this end, the C++ interface to CLIPS does not cache any

information, and before performing any access through the CLIPS external interface it is verified whether the COOL object to be accessed still exists.

5.2 Functional integration

The goal with functional integration is to achieve fine-grain control over the reasoning in the expert system from C++-based components whenever needed. For example, it is sometimes desirable to check whether one single descent from the current skeleton node is possible. Or, the user may prefer to step manually through the tree, getting support from the expert system. The detailed control needed for such tasks has been achieved with a partitioning of the rules into a number of rule modules. Then, the C++ component can set the focus to just the rule system desired and start the reasoning.

Two basic means to interact from the C++ components to the CLIPS expert system exist in the PDE. First, a C++ program can call any of the C functions from the CLIPS external interface. This happens without user visibility in the CLIPS dialog window, and a return value can be obtained from the call. Second, thanks to the graphical user interface written in C++, the C++ program can directly write to the dialog window in such a way that CLIPS thinks it received input at the command line. Using this alternative, no return value can be passed back to the C++ component, but the interaction is visible to the user in the window. It is thus most suited to starting the reasoning or other functions that produce output or a trace in the dialog window.

6 CONCLUSIONS

The realization of the parallel program development environment PDE has successfully achieved the primary goal of making parallel programming as simple as sequential programming within the initial problem domain of stencil-based applications. Moreover, thanks to programming environment support spanning from the design level up to automated code generation and thanks to the reuse of important software components, parallel programming with the PDE can be considered substantially simpler and more reliable than sequential programming in a common procedural language. Apart from the high-level, domain-oriented approach to programming, the PDE offers to the user efficiency preserving portability of software across platforms, reuse of critical software components, and a flexible and comfortable interface.

With a focus on the parts realized using CLIPS, various aspects of the implementation of our knowledge-based parallel program development tool PDE have been shown in this paper. CLIPS in its current version 6.0 [NAS93] proved to have some critical properties making it particularly well-suited for the use within the PDE. Of highest importance are probably the object-oriented capabilities of CLIPS, enabling flexible interfaces to the outside, appropriate representations of knowledge and intermediate problem states, and, together with the CLIPS external interface, a convenient embedding with the C++ components of the PDE. CLIPS is well suited for a rapid prototyping approach to system development, in particular due to the flexibility that the object-oriented mechanism can offer. The PDE development is currently in the fourth prototype. Apart from the first throw-away prototype done in CLOS [BDG+88], each prototype reuses large parts of the previous one, adding completely new components or new functionality.

The problems with the CLIPS language encountered during the PDE development relate in part to the resemblance of CLIPS to CLOS. Examples are the lack of the lists in the sense of Lisp, the static precedence determination for methods, or the inability to pass methods or functions as first-class objects. But alternatives offering similar functionality have been found in all cases. Apart from such CLOS-like items, a suggestion for improvement of the CLIPS language based on our experience is to focus more on object patterns in rules than on facts or templates. An useful extension of the CLIPS external interface, based on the popularity of the C++ programming language, would be the definition and documentation of a C++ frontend for COOL objects.

REFERENCES

[BDG+88] D.G. Bobrow, L.G. DeMichiel, R.P. Gabriel, S.E. Keene, G. Kiczales, and D.A. Moon. *Common Lisp Object System Specification.* X3J13 Document 88-002R, 1988.

[BG94] H. Burkhart and S. Gutzwiller. Steps towards reusability and portability in parallel programming. In K.M. Decker and R.M. Rehmann, editors, *Programming Environments for Massively Parallel Distributed Systems*, pages 147 – 157. Birkhäuser Verlag, Basel, 1994.

[Col89] M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation.* MIT Press, Cambridge, MA., 1989.

[DDR94] K.M. Decker, J.J. Dvorak, and R.M. Rehmann. A knowledge-based scientific parallel programming environment. In K.M. Decker and R.M. Rehmann, editors, *Programming Environments for Massively Parallel Distributed Systems*, pages 127 – 138. Birkhäuser Verlag, Basel, 1994.

[DR92] K.M. Decker and R.M. Rehmann. Simple and efficient programming of parallel distributed systems for computational scientists. Technical Report IAM-92-019, IAM, University of Berne, 1992.

[Dvo93] J. Dvorak. An AI-based approach to massively parallel programming. Technical Report CSCS-TR-93-04, Swiss Scientific Computing Center CSCS, CH-6928 Manno, Switzerland, 1993.

[GR94] J. Giarratano and G. Riley. *Expert Systems: Principles and Programming.* PWS Publishing, Boston, MA., 2nd. edition, 1994.

[NAS93] NASA Lyndon B. Johnson Space Center. *CLIPS Reference Manual*, 6.0 edition, 1993.

[Pol88] C.D. Polychronopoulos. *Parallel Programming and Compilers.* Kluwer Academic Publishers, Boston, 1988.

[ZC90] H. Zima and B. Chapman. *Supercompilers for parallel and vector computers.* Addison-Wesley, Workingham, 1990.

omit

Session 5A: Planning, Oceanographic, and Instruction Applications

Session Chair: Susan Bridges

TRANSPORT AIRCRAFT LOADING AND BALANCING SYSTEM : USING A CLIPS EXPERT SYSTEM FOR MILITARY AIRCRAFT LOAD PLANNING.

J. Richardson, M. Labbé, Y. Belala, V. Leduc
Group of Applied Research in Management Support Systems
Collège militaire royal de St-Jean
Vincent Leduc@CMR.CA

ABSTRACT

The requirement for improving aircraft utilization and responsiveness in airlift operations has been recognized for quite some time by the Canadian Forces. To date, the utilization of scarce airlift resources has been planned mainly through the employment of manpower-intensive manual methods in combination with the expertise of highly qualified personnel. In this paper, we address the problem of facilitating the load planning process for military aircraft cargo planes through the development of a computer-based system. We introduce TALBAS (Transport Aircraft Loading and BAlancing System), a knowledge-based system designed to assist personnel involved in preparing valid load plans for the C130 Hercules aircraft. The main features of this system which are accessible through a convivial graphical user interface, consists of the automatic generation of valid cargo arrangements given a list of items to be transported, the user-definition of load plans and the automatic validation of such load plans.

INTRODUCTION

The aircraft load planning activitiy represents the complex task of finding the best possible cargo arrangement within the aircraft cargo bay, given a list of various items to be transported. In such an arrangement, efficient use of the aircraft space and payload has to be made while giving due consideration to various constraints such as the aircraft centre of gravity and other safety and mission related requirements. The diversity of the equipment to be transported and the variety of situations which may be encountered dictate that highly trained and experienced personnel be employed to achieve valid load plans. Additionally, successful support of military operations requires responsiveness in modifying conceived load plans (sometimes referred to as "chalks") to satisfy last-minute requirements.

The development of a decision support system to assist load planning personnel in their task appears to be a natural approach when addressing the simultaneous satisfaction of the numerous domain constraints. The chief idea behind TALBAS consists of encoding the knowledge of domain experts as a set of production rules. The first section of this paper is devoted to a description of the fundamental characteristics of the load planning problem. A review of major contributions and research work in the field of computer-assisted load planning is then presented. The next section provides a detailed description of the TALBAS architecture and an overview of the modelled reasoning approach used to achieve valid load plans. Finally, a list of possible extensions to the current system is given.

PROBLEM STATEMENT

The problem of interest may be briefly stated as summarized by Bayer and Stackwick [1]:

> " there are various types of aircraft which may be used, each with different capabilities and restrictions and different modes in which it may be loaded, and finally, there is a large variety of equipment to be transported."

Additionally, the load's center-of-balance must fall within a pre-defined Center of Gravity (CG) envelope for all loading methods, otherwise imbalance can render the airplane dynamically unstable in extreme cases.

First addressing the issue of diversity of equipment to be transported, we note that each aircraft can be loaded with cargo including any or all of the following items: wheeled vehicles, palletized cargo, tracked vehicles, helicopters, miscellaneous equipment and personnel. Each type of cargo item requires different considerations. The wide variety of cargo items to be loaded poses a significant difficulty in that a feasible load configuration has to be selected from a very large number of possible cargo arrangements, given a number of pre-defined areas within the aircraft cargo bay which may each impose different restrictions on the type of item which may be loaded depending on weight limitations and other constraints.

Secondly, the load arrangements are dependent upon the following delivery methods : strategic air/land, airdrop, low-altitude parachute extraction system (LAPES) and tactical air/land.
For strategic air/land missions, the emphasis is on maximum use of aircraft space and few tactical considerations are involved. In a tactical air/land mission however, speed and ease with which cargo can be on-loaded and off-loaded takes precedence.

Airdrop operations consist of off-loading equipment and personnel by parachute. LAPES is executed when the aircraft is flown very close to the ground over a flat, clear area. In this case, an extraction parachute is deployed from the rear of the aircraft, after which the aircraft reascends. The equipment, mounted on a special platform, is pulled from the aircraft by the drag on a parachute and skids to a halt. Airdrop and LAPES missions must maintain aircraft balance while dropping cargo and must also provide space between items for parachute cables.

A third issue concerns the characteristics and limitations pertaining to the type of aircraft being loaded. The load capacity of an aircraft depends on its maximum design gross weight (i.e., weight when rounded or upon take-off), its maximum landing gross weight and maximum zero fuel weight (i.e., weight of aircraft and its load when fuel tanks are empty). The maximum design gross weight includes the fuel load to be carried which is determined, at the outset, on the basis of the distance to be flown and other operational requirements. The established quantity of aircraft fuel in turn influences the allowable load which corresponds to the maximum cargo/personnel weight the aircraft can carry.

Finally, constraints pertaining to loading, unloading and in-flight limitations have to be checked when an item is loaded into the aircraft cargo bay. To model a realistic situation, safety and mission related restrictions and parameters such as transport of dangerous goods and cargo/personnel movement priorities, are also taken into account in the set of constraints.

A mathematical formulation of the aircraft loading problem may be obtained through the application of Operations Research techniques. The underlying closely related bin-packing problem (Friesen and Langston [2]) is known to be NP-hard in the strong sense, suggesting it is unlikely that a polynomial solution exists (Garey and Johnson [3]). However, if the requirement of finding the best solution is relaxed to finding a good solution, then heuristic techniques can be applied successfully (Pearl [4]). For a number of reasons to be presented later in this paper, we propose to adopt a heuristic approach based on the knowledge of domain experts. In the following section, we undertake a review of two successful endeavors aimed at facilitating aircraft load planning, that is AALPS and CALM.

The Automated Aircraft Load Planning System (AALPS)

AALPS was developed for the US Forces in the early 1980s, using the Sun workstation as the platform. This load planning tool is a knowledge-based system built using the expertise of highly trained loadmasters.

The underlying system architecture is designed to serve three basic functions: the automatic generation of valid loads, validation of user-defined load plans and user-modification of existing load plans. To perform these tasks, AALPS incorporates four components:

1) a graphical user interface which maintains a graphical image of the aircraft and equipment as cargo items are being loaded;
2) an equipment and aircraft databases which contain both dimensional information and special characteristics such as those required to compute the aircraft CG;
3) the loading module which contains the procedural knowledge used to build feasible aircraft loads; and
4) a database which allows the user to indicate his preference from a set of available loading strategies.

Despite its seemingly powerful features, AALPS would probably necessitate some improvements at significant costs to meet the requirements of the Canadian Forces.

Another initiative in the area of computer-assisted load planning led to the implementation of the Computer Aided Load Manifest (CALM), which was initiated by the US Air Force Logistics Management Centre in 1981.

The Computer Aided Load Manifest (CALM)

The project to develop CALM, formerly called DMES (Deployment Mobility Execution System), was launched at approximately the same time as AALPS but the established objectives were slightly different. In contrast with AALPS, CALM was to be a deployable computer-based load planning system. Consequently, the selected hardware was to be easily transportable and sufficiently resistant for operation in the field.

CALM uses a modified cutting stock heuristic to generate "first-cut" cargo loads, which the planner can alter through interactive graphics.

This system incorporates fewer functional capabilities than AALPS and runs on a PC compatible platform, thus resulting in significantly lower development costs. A major reproach, however, is that this system does not appear to take explicitly into account important load planning data such as the weight of the aircraft fuel being carried, hence leading to incomplete results in some cases. Additionally, the graphical interface would require some improvement to be considered sufficiently user-friendly. In spite of these deficiencies, CALM remains a very useful tool to aid in the load planning process.

In view of the successful research work accomplished in the area of automated aircraft load planning based on the emulation of expert behavior in this field, we have decided to concentrate our efforts on the development of an expert system. The intent in doing so, is to combine the most valuable features present in the existing systems, namely AALPS and CALM, and to adapt them to produce a tool tailored to the Canadian military environment. Our objective is to develop a system with the following features:

1) be deployable to and operable at remote sites;
2) be easy to use and provide a convivial user interface;
3) be capable of handling aircraft load planning problems involving a wide variety of items and several aircraft types used by the Canadian Forces;
4) deal with different types of mission, including the ones with more than one destination;
5) automatically generate valid load plans in a reasonable time;
6) allow the user to alter plans automatically generated by the system;
7) allow users to define their own load plans and issue warnings whenever constraints are violated.

The next section will describe the functional architecture and design of TALBAS which enable the integration of the above described desired features.

THE SYSTEM ARCHITECTURE

The functional architecture depicted at Figure 1 serves the same three basic functions as in AALPS: automatic generation of cargo arrangements, user-definition of load plans and automatic validation of existing load plans. TALBAS consists of an interactive user interface, a loading module, a mission and an aircraft database, and necessary communication links for access to some databases available within the Canadian Forces. The following provides a description of the various databases required to build a feasible cargo load:

1) the aircraft database contains detailed aircraft characteristics including dimensional information and a description of the constraint regions for the C130 Hercules aircraft;
2) the mission database contains detailed mission features mainly in the form of applicable loading strategies;

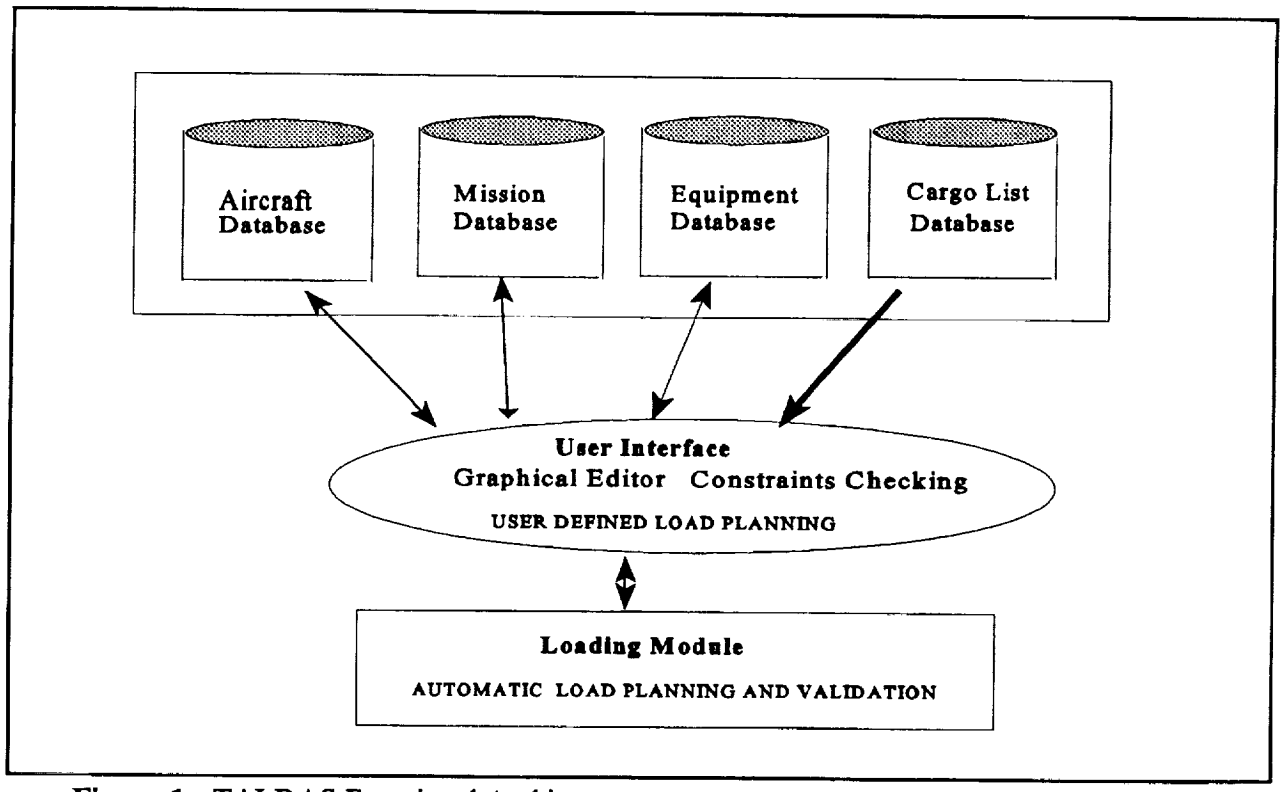


Figure 1 : TALBAS Functional Architecture

3) the equipment and cargo list databases together provide a detailed listing of all cargo items and personnel to be deployed for a mission, including all necessary information about each particular object such as the weight and dimensional characteristics, and the order in which the items are to be transported To minimize the requirement for user input, a set of default values are provided for the majority of the objects' characteristics and these may be changed at the user's request as the situation dictates.

The user interface module performs two main tasks: the graphical display and the constraints checking. The aircraft floor and the items being loaded are graphically represented on the screen. Objects representing loaded cargo items and the various areas of the aircraft cargo floor are drawn on the basis of information available in the different databases described above. All onscreen cargo items are treated as active objects; they can be dragged and moved within the aircraft cargo bay after being selected with a mouse.

The constraints checker ensures that any violation of the center of gravity limits or of other in-flight restrictions is reported through the display of warning message dialog boxes. All the critical load planning data, such as the aircraft CG upon take-off, are computed every time an item is placed or moved within the aircraft cargo area.

The loading module basically consists of an expert system which automatically generates valid load plans. The load planning process will be reviewed in details in the next section.

TALBAS has been developed through the implementation of an incremental prototyping methodology whereby the end user is continuously involved in the refining process of the current prototype. The availability of an ever-improved GUI and loading module, allows for a fail-safe

capture of user requirements. The PC platform has been selected for development purposes in order to produce, in a cost effective fashion, a deployable tool operable at remote sites as a stand-alone system. TALBAS has been designed with an object oriented approach to favor reusability.

The user interface has been implemented using Microsoft Windows and C++. The expert system portion of TALBAS is an enhanced application of the expert system shell CLIPS. This module has been created as a Dynamic-Link Library (DLL). A DLL is a set of functions that are linked with the main application at run time which is, in our case, the user interface. When faced with the problem of having CLIPS communicate with the graphical user interface, three alternatives were contemplated: embedding of CLIPS-based loading module into the main program; implementation of the CLIPS-based loading module as a DLL: or use of the Dynamic Data Exchange (DDE) concept.

DDE is most appropriate for data exchanges that do not require ongoing user interaction. However, the requirement for the devised expert system to constantly monitor any changes made by the user when modifying load plans onscreen has made the approach an unacceptable one. Embedding the CLIPS-based loading module within the main application required that both the user interface and the loading module fit within 64K of static data. This is not possible since the CLIPS-based application (version 6.0) uses all this amount of memory space. On the other hand, implementing the CLIPS-based application as a DLL allows the former and the user interface to be considered separately such that each of the two application codes can fit within the limit of 64K of static data.

The Loading Module

A heuristic approach has been selected as a solution to address the load planning problem. The loading methodologies applied by loadmasters and load planners have been encoded as a set of production rules. This choice was first motivated by the fact that rules of thumb are often the only means available to the experts when seeking a good first try---one requiring the fewest alterations---- to achieve a valid load. As an example, load planning experts generally agree to follow a "60--40'' rule of thumb, namely 60% of the load weight has to be located at the front of the aircraft and 40% at the rear. The second reason for the choice of a knowledge-based system is that the expert knowledge represented in the form of production rules can be easily maintained and updated to incorporate new assets as they are acquired by the Canadian Forces or to adapt to new situations.

Among the three basic objectives, namely the automatic generation of valid load plans, the automatic validation of user-defined load plans and the user-definition of load plans, the first two are achieved using the expert system in TALBAS. The third objective is achieved through the implementation of a graphical user interface allowing the user to manipulate objects onscreen. The expert system is made of two distinct modules as can be seen from Figure 2. The Loading knowledge-base contains all the information related to aircraft loading procedures while the rules contained in the Validation knowledge-base allow the system to recover from situations characterized by a non-balanced aircraft or violated constraints.

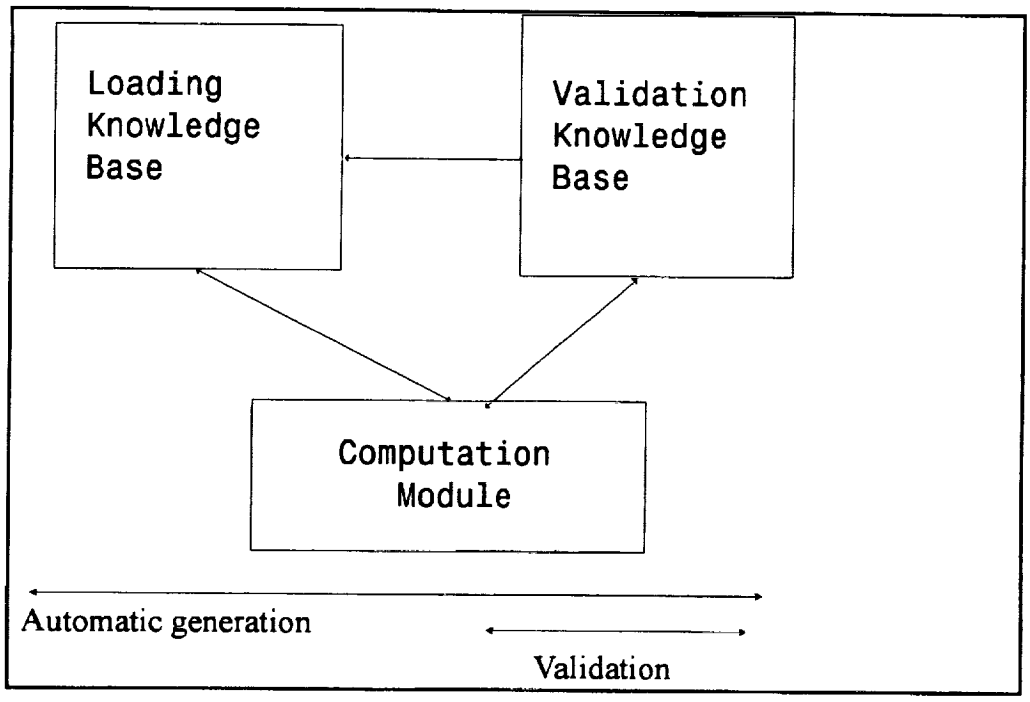


Figure 2 : TALBAS Expert System

The Loading knowledge-base contains all rules which are essential to initially identify one possible load configuration. As in AALPS, an initial cargo state contained in a database of facts is iteratively transformed by our system into an acceptable cargo arrangement by applying a sequence of operators representing the loadmaster's knowledge. In generating a load configuration, pre-defined areas of the aircraft floor are assigned to each category of items or passengers to be transported. For instance, when both cargo and passengers have to be loaded, passengers are usually assigned to the front of the aircraft. Next, a selection of the appropriate item in each category (e.g. vehicles) is made, based on weight, width and length considerations. Since the aircraft balance is a primary concern, a preliminary selection among all cargo items is accomplished on the basis of weight followed by consideration of dimensional information depending on the type of item selected. The integrated computation module will subsequently compute the achieved CG and verify that numerical constraints have not been violated.

At this step, the coded heuristics contained in the loading knowledge-base are likely to have produced an acceptable "first-cut" load plan. However, the system will try to find a cargo arrangement characterized by an optimal CG value while ensuring that all constraints are satisfied. If the initial load is not acceptable, the system will make use of rules stored in the validation knowledge-base to first slide the loaded items, if there is sufficient aircraft space remaining to do so, or attempt to rearrange these items if sliding operations are not feasible. TALBAS will stop when it has identified either an optimal load plan or an acceptable one. In such cases where no possible solutions may be found for the list of given items to be loaded and no more permutations

can be made, the system issues a warning message indicating that no valid load plans can be found with that combination of items.

A second role played by the expert system module concerns the validation of user-defined load plans. In this case, it operates in the same fashion as described above when automatically generating valid load plans with the difference that the initial load plan is produced by the user himself.

CONCLUSION AND FUTURE WORK

In this paper, we have presented a knowledge-based alternative to facilitate load planning of military aircraft. We have successfully incorporated most of the valuable features present in the existing systems, AALPS and CALM, and adapted them to produce a tool tailored to meet the specific requirements of the Canadian military environment. The major concerns which have been addressed were the deployability and conviviality of the designed system. The CLIPS-based expert system module automatically generates valid load configurations and validates user-defined/modified load plans. The developed graphical user interface allows for the easy alteration of existing plans.

Efforts in the design of the developed system have been primarily focused on the loading of the C130 Hercules aircraft, since it is currently the principal airlift resource used by the Canadian Forces for the transport of cargo and personnel. The system may however, be easily adapted to accommodate other existing or future types of Canadian Forces transport aircraft. Expansion of TALBAS to permit the loading of several aircraft, while giving due consideration to the movement priority level assigned to each item to be airlifted, is currently under development.

REFERENCES

[1] Bayer, W.K., and Stackwick, D.E., "Airload Planning Made Easy", ARMY LOGISTICIAN, May-June, 1984, pp. 34-37.

[2] Friesen, D.K., and Langston, M.A., "Bin Packing: On Optimizing the Number of Pieces Packed", BIT, 1987, pp. 148-156.

[3] Garey, M.R., and Johnson, D.S., COMPUTERS AND INTRACTABILITY: A GUIDE TO THE THEORY OF NP-COMPLETENESS, W.H. Freeman, San Francisco, California, 1979.

[4] Pearl, J., HEURISTICS: INTELLIGENT SEARCH STRATEGIES FOR COMPUTER PROBLEM SOLVING, Addison-Wesley, Reading, Massachusets, 1984.

[5] Anderson, D., and Ortiz, C., "AALPS: A Knowledge-Based System for Aircraft Loading", IEEE, Winter 1987, pp. 71-79.

[6] Cochard, D.D., and Yost, K.A., "Improving Utilization of Air Force Cargo Aircraft", INTERFACES, January-February, 1985, pp. 53-68.

[7] Ignizio, J.P., INTRODUCTION TO EXPERT SYSTEM: THE DEVELOPMENT AND IMPLEMENTATION OF RULE-BASED EXPERT SYSTEMS, McGraw-Hill, Houston, Texas, 1991.

[8] Ng, K.Y.K., "A Multicriteria Optimization Approach to Aircraft Loading", OPERATIONS RESEARCH, November-December, 1992, pp. 1200-1205.

PREDICTING AND EXPLAINING THE MOVEMENT OF MESOSCALE OCEANOGRAPHIC FEATURES USING CLIPS

Susan Bridges
Liang-Chun Chen
Computer Science Department
Mississippi State University[1]

Matthew Lybanon
Naval Research Laboratory
Stennis Space Center, MS

ABSTRACT

The Naval Research Laboratory has developed an oceanographic expert system that describes the evolution of mesoscale features in the Gulf Stream region of the northwest Atlantic Ocean. These features include the Gulf Stream current and the warm and cold core eddies associated with the Gulf Stream. An explanation capability was added to the eddy prediction component of the expert system in order to allow the system to justify the reasoning process it uses to make predictions. The eddy prediction and explanation components of the system have recently been redesigned and translated from OPS83 to C and CLIPS and the new system is called WATE (Where Are Those Eddies). The new design has improved the system's readability, understandability and maintainability and will also allow the system to be incorporated into the Semi-Automated Mesoscale Analysis System which will eventually be embedded into the Navy's Tactical Environmental Support System, Third Generation, TESS(3).

1. INTRODUCTION

One of the major reasons CLIPS is so widely used is the ease with which it allows a rule base to be incorporated as one component of a larger system. This has certainly been the case with the eddy prediction component of the Semi-Automated Mesoscale Analysis System (SAMAS) (3). SAMAS is an image analysis system developed by the Naval Research Laboratory that includes a variety of image analysis tools that enable the detection of mesoscale oceanographic features in satellite images. Unfortunately, in the North Atlantic, many of the images are obscured by cloud cover for lengthy periods of time. A hybrid system for use when features cannot be detected in images has been developed that consists of a neural network that predicts movement of the Gulf Stream and a rule base that predicts movement of eddies associated with the Gulf Stream. The Gulf Stream and eddy prediction components were both originally implemented in OPS83 (4). The Gulf Stream Prediction Module has been replaced by a neural network (3) and an explanation component has recently been added to the OPS83 version of the eddy prediction component (1). The eddy prediction rule base, WATE (Where Are Those Eddies), has been translated to CLIPS because of the ease of integrating a CLIPS rule base into a larger system, the ability to access routines written in C from CLIPS rules, and the support CLIPS provides for the forward chaining reasoning used by the eddy prediction system. The explanation component of WATE uses meta rules written in CLIPS to compose either rule traces or summary explanations of the predicted movement of eddies.

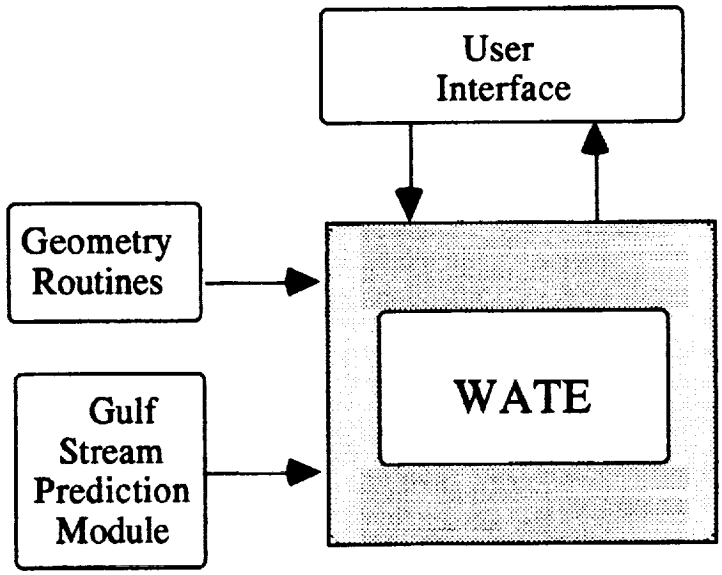
[1]This work was supported by the Naval Research Laboratory, Stennis Space Center under contract NAS 13-330.

2. SYSTEM ARCHITECTURE

2.1 External Interfaces

The WATE component interacts with other SAMAS components as shown in Figure 1. WATE interacts with the User Interface in two ways. First, WATE is invoked from the User Interface when the user requests a prediction of Gulf Stream and eddy movement for a specified time. Second, as WATE predicts the movement of the Gulf Stream by calling the Gulf Stream Prediction Module and eddies by running the rule base, WATE calls User Interface routines to update the graphical display of the positions of the Gulf Stream and eddies. The eddy prediction rules call the Geometry Routines to compute distances and locations. WATE invokes the Gulf Stream Prediction Module to predict the movement of the Gulf Stream for each time step. The position of the Gulf Stream predicted by the neural network component must be accessed by the eddy prediction rule base since the movement of eddies is influenced by the position of the Gulf Stream.

Figure 1. External Interfaces of WATE



2.2 Redesigned Control Structure

The control structure for the original expert system was written in procedural OPS code and had been modified a number of times as the graphical user interface, eddy prediction, Gulf Stream prediction, and explanation components were either modified or added. The result was a control structure that was not modular and that contained a substantial number of obsolete variables and statements. When the system was converted from OPS83, the control structure was completely redesigned and rewritten in C. Pseudo code for the redesigned control structure is shown in Figure 2. The resulting code was more efficient because it was written in compiled C code rather than interpreted OPS83 procedural code.

Figure 2. Control Structure of WATE

```
Initialization
        GetUserOptions
        SetUpCLIPS
        SetUpPVWave
        SetUpGulfStream

Prediction
        Display initial positions of GS and eddies
        for each time step do
                update time
                MoveGulfStream
                Update display of GS
                MoveEddies
                Update display of eddies

Explanation
        Explanation Composition
        Explanation Presentation

FinalOutput
```

2.3 Translation from OPS83 to CLIPS

The original OPS83 working memory elements and rules had been completely restructured to support an explanation component (1). The translation of this restructured OPS83 code into CLIPS was fairly simple since CLIPS has evolved from the OPS line. There is a very straightforward translation from OPS83 working memory elements to CLIPS fact templates and from OPS rules to CLIPS rules. In some cases, the OPS83 rules called OPS83 functions or procedures. These functions and procedures were translated to C.

3. EXPLANATION COMPONENT

The explanation component allows the user to ask for either a rule trace or summary explanation for the prediction of the movement of each eddy at the end of each prediction cycle. The rule trace explanations give a detailed trace of the instantiation of all rules that were fired to predict the movement of an eddy. Although this type of trace has proven to be very useful in debugging the system, it was immediately apparent that it contained a great deal of information that would be of little interest to most users. Interviews with domain experts were used to determine the information that would be of most interest to a user. The types of information they identified was used to design the summary explanations. Presentation of these explanations requires that the line of reasoning of the system be captured as the rules fired and that information from this rule trace be extracted and organized for presentation to the user.

3.1 Rule Firing Capture

Capturing the rule trace for this domain in a usable form is simplified because all explanations (both trace and summary) are focused on a particular eddy. This means that all of the

rule-firings pertaining to the movement of one eddy can be stored together and presented as one explanation. This is accomplished by asserting a *rule-fire-record* template fact for each eddy for each time step with the following deftemplate definition:

```
(deftemplate rule-fire-record
    (field       ringtype      (allowed-symbols wcr ccr))
    (field       refno         (type INTEGER))
    (field       time          (type INTEGER))
    (multifield  rules-fired   (type SYMBOL)))
```

The ringtype, eddy identifier (*refno*), and time stamp (*time*) uniquely identify each rule-fire-record. The *rules-fired* multifield is used to store a list of the names of the rules that fired to predict the movement of the eddy during this time step. Each time a rule fires as part of the prediction process for a particular eddy, the rule name is added to the end of the *rules-fired* list .

A second set of template facts is used to record the instantiation of each rule that fires. Each time a rule fires, a *values-for-explanation* template fact is asserted which gives the value bound to each variable when the rule was fired. The deftemplate definition for *values-for-explanation* is:

```
(deftemplate values-for-explanation
    (field       rule-name     (type SYMBOL))
    (field       ringtype      (allowed-symbols wcr ccr))
    (field       refno         (type INTEGER))
    (field       time          (type INTEGER))
    (multifield  var-val       (type SYMBOL)))
```

This template contains slots for the rule name, the eddy identifier and type, and the time stamp. In addition, it contains a multifield slot whose value is a sequence of variable value pairs that gives the name of each variable used in the rule-firing and the value bound to that variable when the rule fired. This approach can be used in this domain because a single rule will never fire more than one time for a particular eddy during one time step, and all slots in templates used by the eddy prediction rules are single value. The records of the rules that fired for a particular eddy are used by meta rules to produce the explanations.

3.2 Explanation Construction and Presentation

If the user requests an explanation for a specific eddy, a set of explanation meta rules are used to construct an explanation for the predicted movement of an eddy. The user may request either a rule-trace explanation or a summary explanation. When the user makes the request, a sequence of *explain-eddy* facts for that eddy are asserted each with a progressively higher time stamp. The fact template for an *explain-eddy* fact is:

```
(explain-eddy  ccr | wcr  <ref-no>  <time>  summary | rule-trace).
```

The presence of this fact causes the *explain-single-eddy* rule to fire one time for each rule that fired to predict the movement of the eddy for that time step. Each *explain-single-eddy* rule firing matches a rule name from the *rules-fired* slot of the *rule-fire-record* with a *values-for-explanation* template for that eddy type, number, and time stamp. A *fill-template* fact is then asserted into working memory which contains all of the information needed to explain that rule firing--either in rule-trace or summary form. For each rule, there are two explanation meta rules. The first is used when a rule-trace has been requested and is a natural language translation of the rule. It will give the value of all variables used in the rule instantiation. The second is used when a summary has

been requested. It gives a much shorter summary of the actions of the rule. A few rules that are used to control the sequence of rule-firing produce no text for a summary explanation. When an explanation metarule fires, it causes a natural language translation of the rule to be sent to the user interface for presentation to the user.

The user may request an explanation of the movement of all eddies instead of just a single eddy. In this case, the process above is simply repeated for each eddy.

4. SUMMARY AND FUTURE WORK

WATE has been successfully converted from OPS83 to C and CLIPS. This conversion will facilitate the incorporation of WATE into SAMAS 1.2 which will eventually be embedded in TESS(3). The modular control structure of WATE is easier to understand and maintain than that of the previous system. The explanation component has been implemented using CLIPS metarules. This causes some additional maintenance burden since the two metarules that correspond to each rule must be modified if a rule is modified. In the present system, the *rule-fire-record* and *values-for-explanation* template facts are asserted by each individual rule. We are currently modifying the CLIPS inference engine to capture this information automatically as the rules fire.

Explanations produced by the current system have two major shortcomings. First, there is still a great deal of room for improvement in the summarization capabilities of the system. In particular, the system should be summarizing over both temporal and spatial dimensions. If an eddy's predicted movement is essentially in the same direction and speed for each time step, then all of this information should be collapsed into one explanation. Likewise, if several eddies all have similar movement over one or more time steps, this should be collapsed into a single explanation. The second shortcoming deals with the lack of explanation of the predictions of the neural network component. Some recent results reported in the literature have addressed this sort of problem.

5. REFERENCES

1. Bridges, S. M., and Lybanon, M. "Adding explanation capability to a knowledge-based system: A case study," pp. 40-49, Applications of Artificial Intelligence 1993: Knowledge-Based System in Aerospace and Industry, Orlando, FL, April, 1993.

2. Lybanon, M. "Oceanographic Expert System: Potential for TESS(3) Applications," Technical Note 286, Naval Oceanographic and Atmospheric Research Laboratory, Stennis Space Center, MS, 1992.

3. Peckinpaugh, S. H. Documentation for the Semi-Automated Mesoscale Analysis System 1.2, 1993.

4. Thomason, M. G., and Blake, R. E. NORDA Report 148, Development of An Expert System for Interpretation of Oceanographic Images. Naval Ocean Research and Development Activity, Stennis Space Center, MS, 1986.

KNOWLEDGE BASED TRANSLATION AND PROBLEM SOLVING IN AN INTELLIGENT INDIVIDUALIZED INSTRUCTION SYSTEM

Namho Jung and John E. Biegel
Intelligent Simulation Laboratory
Department of Industrial Engineering and Management Science
University of Central Florida
Orlando, Florida 32816

namho@oak.ists.engr.ucf.edu
biegel@oak.ists.engr.ucf.edu

ABSTRACT

An Intelligent Individualized Instruction (I^3) system is being built to provide computerized instruction. We present the roles of a translator and a problem solver in an intelligent computer system. The modular design of the system provides for easier development and allows for future expansion and maintenance. CLIPS modules and classes are utilized for the purpose of the modular design and inter module communications. CLIPS facts and rules are used to represent the system components and the knowledge base. CLIPS provides an inferencing mechanism to allow the I^3 system to solve problems presented to it in English.

INTRODUCTION

The Intelligent Individualized Instruction (I^3) system is an intelligent teaching system that makes possible the knowledge transfer from a human to a computer system (knowledge acquisition), and from the system to a human (intelligent tutoring system (ITS)). The ITS portion of the I^3 system provides an interactive learning environment where the system provides self-sufficient instruction and solves the problem presented in a written natural language. Self-sufficient instruction means that no instructor is required during the learning cycle. Solving problems written in a natural language means that the system is able to 'understand' the natural language: It is not an easy task, especially without any restriction on the usage of vocabulary and/or format.

Two I^3 system modules, a Translator and an Expert (a problem solver and a domain glossary), understand a problem presented to it in English by translation and keyword pattern matching processes. The I^3's pattern matching method uses the case-based parsing [Riesbeck and Schank 1989] that searches its memory (knowledge base) to match the problem with stored phrase templates. Unlike other case-based parsers (e.g., CYC project [Lenat and Feigenbaum, 1989]) the I^3 system does not understand the problem statement as a human does. A human uses common sense or other background knowledge to understand it. Rather the I^3 system 'understands' enough about the problem for the system to be able to solve the problem. We will discuss how the system 'understands' the human language.

AN INTELLIGENT INDIVIDUALIZED INSTRUCTION (I^3) SYSTEM

The I^3 system is a Knowledge Based System that is composed of domain dependent modules, domain independent modules, and a User interface module. The domain dependent modules (the Domain Expert and the Domain Expert Instructor) carry the domain expertise that enables the other modules remain domain independent. The separation of these domain dependent modules from the rest of the system makes the system reusable. Whenever the I^3 system is applied to another domain, only the domain-dependent knowledge of the new domain is needed.

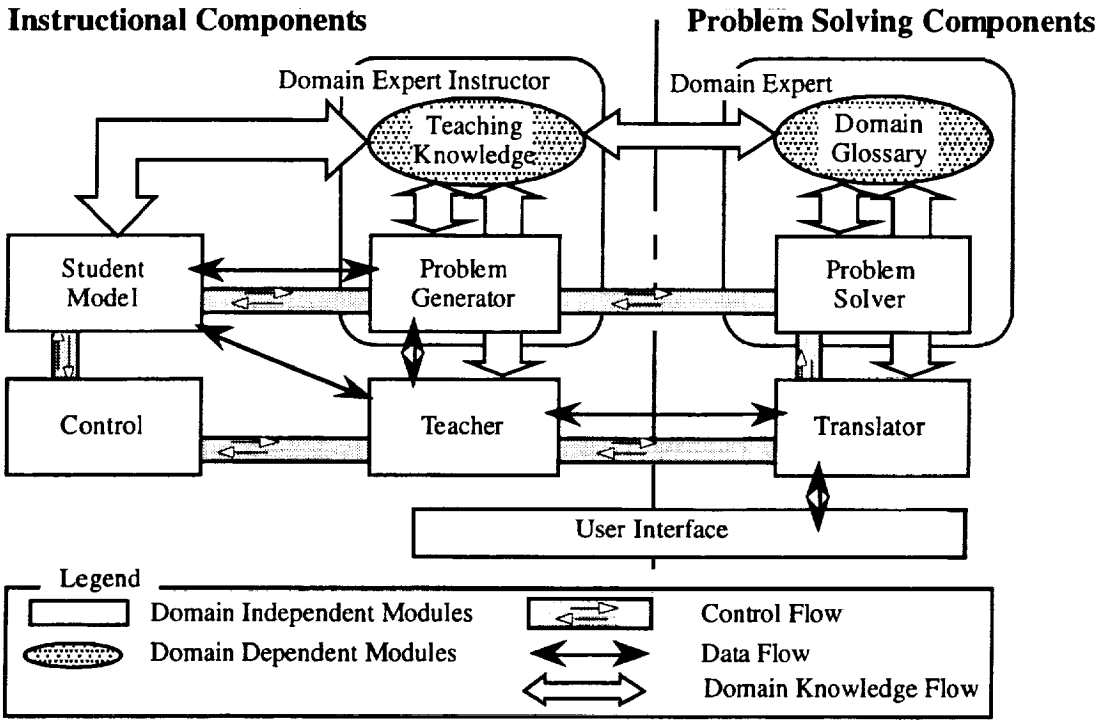


Figure 1. System architecture of I³'s intelligent teaching system

The goal of the I³ system is to provide a student with individualized learning to attain competency [Biegel 1993]. The I³ knowledge presentation subsystem generates self-sufficient instructions. The I³ system contains instructional components (Student Model, Teacher, Domain Expert Instructor, Control, and User Interface) and problem solving components (Translator and Domain Expert).

- The Student Model module evaluates and maintains the trainee's performance overall or on individual lessons.
- The Teacher module contains the knowledge about generic didactic strategies to customize each lesson by selecting and sequencing the instruction material.
- The Domain Expert Instructor (DEI) module represents the domain dependent teaching methodology. The DEI module provides the Teacher module with the teaching strategies, and the Student Model module with the evaluation criteria for the individual and/or overall lesson.
- The Control module manages the operation and maintains the modules' communications within the system.
- The User Interface module handles all communications between the user and the system.
- The Translator module parses a trainee's input and translates it into a system-understandable format.
- The Domain Expert (DE) module contains a knowledge base representing the problem-solving knowledge of a human domain expert.

DESIGN OF THE TRANSLATOR AND THE PROBLEM SOLVER

The Translator module and the Domain Expert module provide the user with the proper interpretation of the problem and the correct solution. Together, they allow a system to apply domain expert heuristics to solve problems by pattern matching. Pattern matching allows the

system to 'understand' a problem statement within the domain for which the system has been built. The problem statement can be translated into a set of rules and facts. Since the problem solver cannot translate English (or other natural language) directly into computer readable code, it relies on a translator to provide a communication mechanism between the user and the computer.

The Translator provides a full duplex communication medium between the human and the computer. The user will not be constrained in the format of the input problem and will not be required to do any parsing or be restricted to a limited syntax. The Translator module translates the domain jargon of an input problem into a system understandable format.

The major tasks of the translator are: (1) to convert text into computer readable code, and (2) to provide a knowledge base conversion and problem representation process. The text conversion process includes checking for correct spelling and removing all unnecessary symbols. The knowledge base conversion process includes (1) the conversion of a written number to a numeric value, (2) filtering out unnecessary words, and (3) replacing words with stem (root) or exemplar words. The knowledge based process uses a number conversion list, a list of words unnecessary for problem solving, and a domain thesaurus. The Intelligent Individualized Instruction system separates the knowledge base (domain glossary) which is the collection of the domain thesaurus, the domain template dictionary, and the unnecessary word listing, from the translation process.

The Domain Expert interprets the translated input using a domain vocabulary as a reference, selects a suitable method from a list of solution methods, and finds a solution. The Domain Expert (DE) consists of three parts, a Domain Glossary (DG), a set of Managers, and a Problem Solver (PS). The DE imitates the expert's methodologies of problem solving: The DG acts as the expert's memory by providing the necessary problem solving knowledge. Each Manager acts as the expert's procedural knowledge by solving a routine of the problem in one specific area. The PS acts as the scheduling and reasoning processes by controlling and scheduling the Managers in proceeding toward the solution of a problem.

The Domain Glossary represents the domain expertise. The DG consists of a domain template dictionary, a domain thesaurus, an unnecessary word listing, a domain symbol list, and a domain theory list. The DG represents relational knowledge (e.g., one year is twelve months), factual knowledge (e.g., 'interest rate' means domain variable i), and a list of words and symbols (e.g., '%' has a special meaning of interest rate in the Engineering Economy domain).

Special dictionaries for the domain provide benefits such as faster look up access than a general dictionary, and a higher priority to find the correct interpretation. The problem solver does not have to consider all different combinations of words' variables. It searches its own smaller dictionary that contains the necessary information in the application domain. English text (problem statement) is interpreted/translated by using the domain thesaurus without knowing the general meaning of the word. The thesaurus contains all words relevant to the domain. Each word is connected to a list of possible interpretations. Each word in the statement is looked up in the thesaurus and replaced by all relevant symbols.

The number of words in the domain vocabulary will vary between domains, but a vocabulary of 500 or so words and symbols will most likely cover most of the undergraduate engineering domains.

Each Manager handles one specific area of problem solving. It is a self-sufficient object that contains procedural knowledge (rules) and factual knowledge (facts) attached to it, and that knows how to handle situations upon request. When activated, a Manager searches the input statement for matched patterns in its templates. If a match is found, the Manager processes or interprets that portion of the problem statement. For example, a Date Manager knows how to interpret key terms that are related to the date, how to compare two date instances, and how to calculate the period

from two date instances. When a problem statement contains "... invest $5000 on January 1, 1994, ... will be accumulated in 5 years hence ... ", the Date Manager replaces the statement with " ... invest $5000 on [D1] ... will be accumulated in [D2] ... " where [D1] and [D2] are instances of a Date class and are represented as:

```
([Date::D1] is a Date                ([Date::D2] is a Date
    (year        1994)                   (year        5)
    (month  1)                           (month  0)
    (day         1)                      (day         0)
    (base        none) )                 (base        [Date::D1]) )
```

Some mangers handle both domain dependent and independent situations based on the factual knowledge they have. Communication among managers can be made through dedicated communication channels, such as CLIPS class objects or templates.

The Domain Expert module is of a modular design and maintains the separation of strategic knowledge from factual knowledge. The domain expertise can be categorized into three levels: high level control knowledge (a Problem Solver), middle level procedural knowledge (Managers), and low level factual knowledge (a Domain Glossary). By nature, the low level factual knowledge tends to be domain specific, and the high level control knowledge tends to be a domain independent. Any addition to the knowledge base can be accomplished by adding a Manager and its associated knowledge into the DG.

PROBLEM SOLVING IN THE I^3 SYSTEM

The problem solver applies a separate-and-solve method that breaks a problem statement into several small blocks, interprets each block, and then logically restructures them. The problem solving steps include interpretation of the problem statement, selection of the formula, and mathematical calculation. The steps are depicted in Figure 2 in which boxes on the left hand side represent the changes of the problem statement from input English text to the answer. The middle ovals show the problem solving processors. The right hand side boxes represent the domain expertise of the domain glossary. The problem solving process is generic so it can be used in other domains if the new domain expertise is available.

The I^3 system problem solving routine is performed by the problem solving components: the User Interface, the Translator, and the Domain Expert (the Problem Solver, the Managers, and the Domain Glossary). The routine includes initial domain independent processes (translating and filtering an input problem), and main domain dependent processes (interpreting the problem, selecting a solution method, and deriving an answer).

A user enters an engineering economics problem through the User Interface, as shown in Figure 3. The Translator performs filtering process by checking correct spelling using its English dictionary. The Translator converts the input problem statement into system understandable format; plural words to singular; past tense to present; uppercase words to lowercase; verbal numbers to numeric values; and separates symbols from numeric values (Figure 4). For example, part of the problem statement "If *$10,000* is *invested* at 6% interest *compounded* annually" becomes "if $ 10000 is invest at 6 % interest compound annual". Now, all the elements in the problem statement are known to the system.

The problem statement is divided into several blocks in order to distribute the complexity of the problem (Figure 5). Each block is a knowledge unit that contains a domain variable and a numeric value. The knowledge unit contains necessary as well as unnecessary information for interpreting the problem statement. Any unnecessary word such as 'at' must be removed before reasoning, because they only required overhead on the system during the process of reasoning. As an

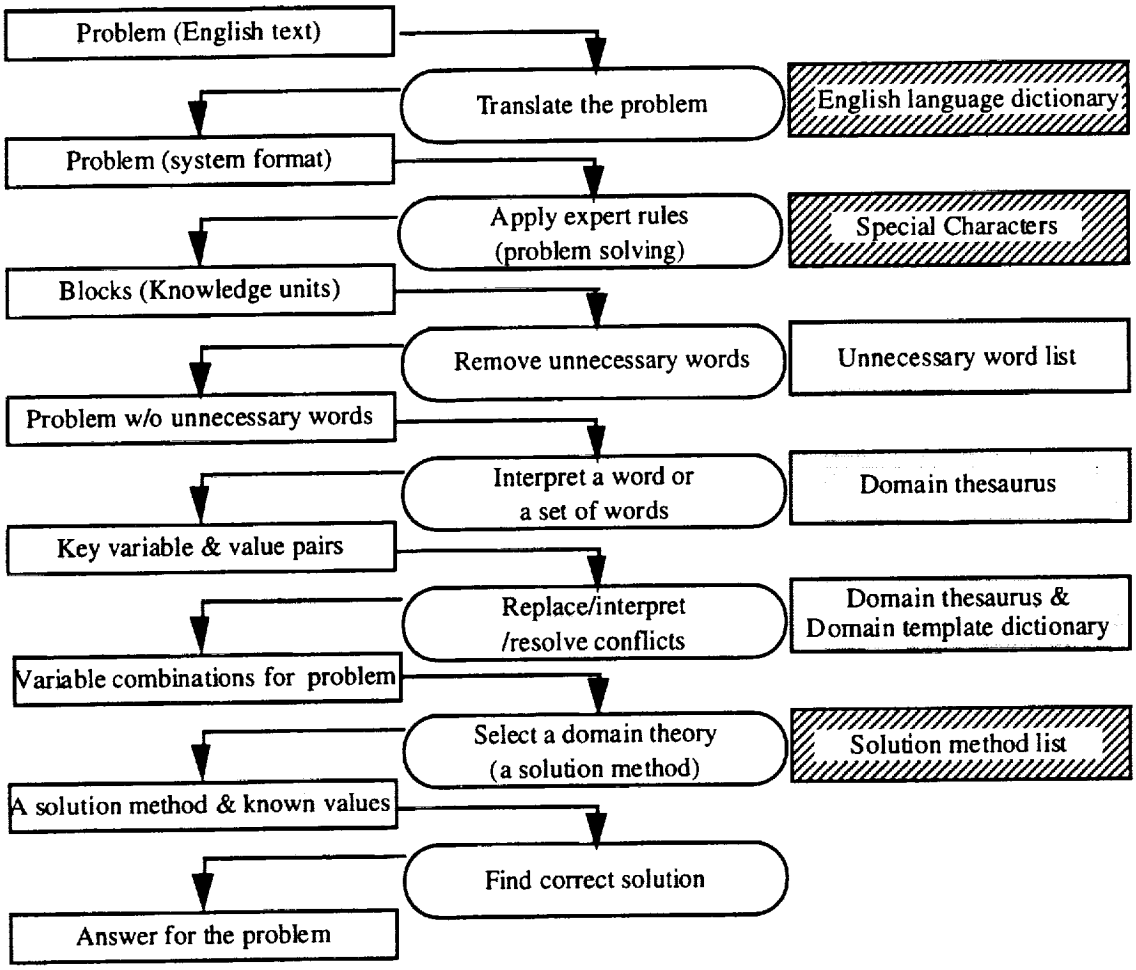


Figure 2. problem solving process

How much money will you have accumulated three years from now if $10,000 is invested at 6% interest compounded annually?

Figure 3. A sample problem in engineering economics

Plural to singular:	years => year
past tense to present tense:	invested => invest compounded => compound accumulated => accumulate
Upper case to lower case:	How => how
verbal number to numeric number:	three => 3
remove comma within a number:	10,000 => 10000
separate symbol from number:	$10000 => $ 10000 6% => 6 %

Figure 4. Conversion process

Block (Knowledge unit)	Unnecessary word
1. how much money will you have accumulate	you have
2. three year from now	
3. if $10,000 is invest	if, is
4. at 6% interest compound annually	at

Figure 5. Removing unnecessary words from each knowledge unit

Knowledge unit	Interpretation
1. how much money will ... accumulate	Find F
2. 3 year from now	N = 3
3. ... $ 10000 ... invest	P = 10000
4. ... 6 % interest compound annual	i = 6%

Figure 6. Knowledge Units of the Sample Problem

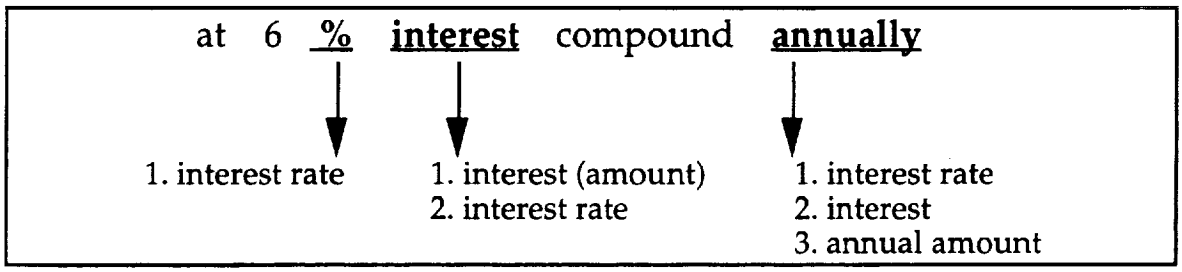


Figure 7. Domain Thesaurus Interpretation Example

Given P, i, N, and Find F.
Solution strategy is F = P (F / P, i%, N))

Figure 8. Finding a Solution Strategy

instance, a block "if $10000 is invest" will be interpreted as a present worth "P = $10000" because 'if' and 'is' are unnecessary, 'invest' is used previously as present worth, and '$10000' is a value of the variable. Unnecessary words can be found in all problems in the domain, but not in the list of domain templates. A domain template is a sequence of words, a knowledge unit, that is used to interpret a domain variable. The unnecessary word is not used uniquely: it could be found in the templates for all different variables.

The Problem Solver interprets each knowledge unit by applying the domain thesaurus and domain template dictionary. For example, when a text block, 'at 6% interest compound annual' is given to the system, the knowledge base provides interpretation of the block: 1) word by word: The word 'at' is an unnecessary word for solving the problem. Next word, '%', will be interpreted as 'interest rate,' and 'compound' as 'interest rate.' The word 'interest' has two meanings: 'interest' (amount of money) and 'interest rate' (rate). The last word 'annual' could be represented in three different variables: 'interest,' 'interest rate,' and 'annual value' (Figure 7). 2) as a template 'interest compound annual' meaning 'interest rate.' Such conflicts will be resolved by selecting an interpretation with the highest priority among all different possibilities. The knowledge base provides the necessary knowledge to determine which one has higher priority.

The Problem Solver sends the interpretation of the problem statement to the domain theory selector. The interpretation of the problem (for example, P = $10000, i = 6%, N = 3 year, and F is unknown) is used to select an appropriate solution method (F = P (F / P, i%, N)) (Figure 8). The system applies the interpretation of the problem to the solution method (F = 10000 (F / P, 6%, 3)). The solution found is presented to the user through the User Interface.

CONCLUSION

The Translator and the Problem Solver in the I^3 system have demonstrated that the knowledge based interpretation of natural language is feasible. Modular design of the Problem Solver provides the system's expandability and reusability. Expanded problem solving capability of the system can be accomplished by adding more knowledge to the Domain Glossary. Reusability can be enhanced by replacing or adding managers to the Problem Solver without reprogramming other parts of the system. Combining rule based processing with objects (or an integration of object oriented system with an intelligent system) makes it possible to define domain knowledge about the application further than with rules alone.

The I^3 system is being developed on an IBM compatible 486 machine using the C/C++ programming language (Microsoft Visual C++) and CLIPS 6 (C Language Integrated Production System, by NASA Lyndon B. Johnson Space Center, a forward chaining expert system shell based on the Rete algorithm).

REFERENCE

1. Biegel, John, "I^3: Intelligent Individualized Instruction," PROCEEDINGS OF THE 1993 INTERNATIONAL SIMULATION CONFERENCE, San Francisco, CA., OMNIPRESS, Madison, Wisconsin, November, 1993, 243-249.

2. Chung, Minhwa, and Moldovan, Dan, "Applying Parallel Processing to Natural-Language Processing." IEEE EXPERT INTELLIGENT SYSTEMS & THEIR APPLICATIONS, IEEE Computer Society, Vol. 9, Number 1, February, 1994, 36-44.

3. Martin, Charles, "Case-based Parsing," INSIDE CASE-BASED REASONING, Riesbeck Christopher K. and Schank, Roger C., Lawrence Erlbaum Associates, Publishers, Hillsdale, New Jersey, 1989, 319-352.

4. Lenat, Douglas, and Feigenbaum, Edward, "On the Thresholds of Knowledge," APPLICATIONS OF EXPERT SYSTEMS, Vol. 2, Ed. Quinlan, J. Ross, Addison-Wesley Publishing Company, Sydney, Australia, 1989, 36-75.

omit

Session 5B: Debugging, Optimization, and Prototyping Extensions

Session Chair: Steve Scott

MIRO: A DEBUGGING TOOL FOR CLIPS INCORPORATING HISTORICAL RETE NETWORKS

Sharon M. Tuttle
Management Information Systems, University of St. Thomas
3800 Montrose Blvd., Houston, TX, 77006

Christoph F. Eick
Department of Computer Science, University of Houston
Houston, TX 77204-3475

ABSTRACT

At the last CLIPS conference, we discussed our ideas for adding a temporal dimension to the Rete network used to implement CLIPS. The resulting historical Rete network could then be used to store 'historical' information about a run of a CLIPS program, to aid in debugging. MIRO, a debugging tool for CLIPS built on top of CLIPS, incorporates such a historical Rete network and uses it to support its prototype question-answering capability. By enabling CLIPS users to directly ask debugging-related questions about the history of a program run, we hope to reduce the amount of single-stepping and program tracing required to debug a CLIPS program. In this paper, we briefly describe MIRO's architecture and implementation, and the current question-types that MIRO supports. These question-types are further illustrated using an example, and the benefits of the debugging tool are discussed. We also present empirical results that measure the run-time and partial storage overhead of MIRO, and discuss how MIRO may also be used to study various efficiency aspects of CLIPS programs.

1. INTRODUCTION

In debugging programs written in a forward-chaining, data-driven language such as CLIPS, programmers often have need for certain *historical* details from a program run: for example, when a particular rule fired, or when a particular fact was in working memory. In a paper presented at the last CLIPS conference [4], we proposed modifying the Rete network, used for determining which rules are eligible to fire at a given time, within CLIPS, to retain such historical information. The information thus saved using this *historical Rete network* would be used to support a debugging-oriented question-answering system.

Since the presentation of that paper, we have implemented historical Rete and a prototype question-answering system within MIRO, a debugging tool for CLIPS built on top of CLIPS. MIRO's question-answering system can make it much less tedious to obtain historical details of a CLIPS program run as compared to such current practices as rerunning the program one step at a time, or studying traces of the program. In addition, it turns out that MIRO may also make it easier to analyze certain efficiency-related aspects of CLIPS program runs: for example, one can much more easily determine the number of matches that occurred for a particular left-hand-side condition in a rule, or even the number of matches for a subset of left-hand-side conditions (those involved in *beta memories* within the Rete network).

The rest of the paper is organized as follows. Section two briefly describes MIRO's architecture and implementation. Section three then gives the currently-supported question-types, and illustrates how some of those question-types can be used to help with debugging. Empirical results regarding MIRO's run-time and partial storage overhead costs are given in section four, and section five discusses some ideas for how MIRO might be used to study various efficiency aspects of CLIPS programs. Finally, section six concludes the paper.

2. MIRO'S ARCHITECTURE AND IMPLEMENTATION

To improve CLIPS' debugging environment, MIRO adds to CLIPS a question-answering system able to answer questions about the current CLIPS program run. We used CLIPS 5.0 as MIRO's basis. Figure 1 depicts the architecture of MIRO. Because questions useful for debugging will often refer to

historical details of a program run, MIRO extends the CLIPS 5.0 inference engine to maintain historical information about the facts and instantiations stored in the working memory, and about the changes to the agenda. Moreover, in order to answer question-types we provided query-operators that facilitate answering questions concerning past facts and rule-instantiations, and an agenda reconstruction algorithm that reconstructs conflict-resolution information from a particular point of time.

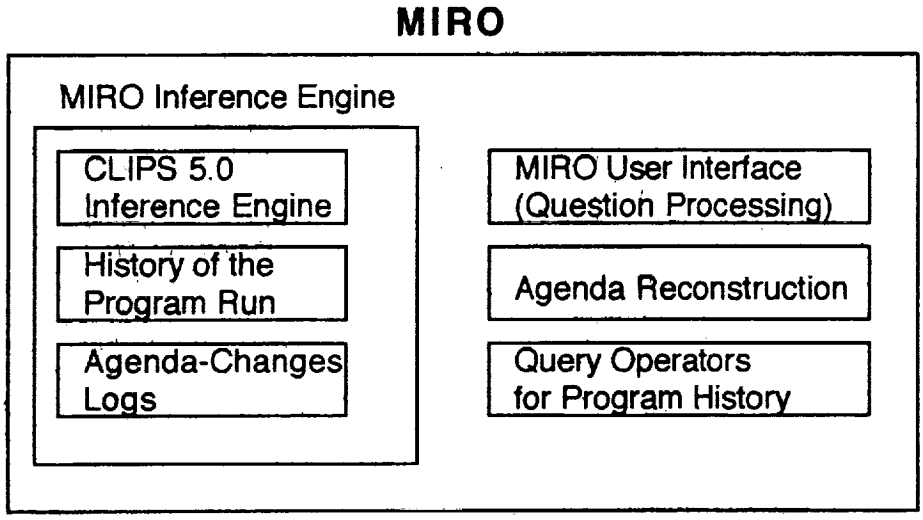


Figure 1 - The MIRO Debugging Environment

One could describe MIRO as a tool for helping programmers to analyze a program run; it assists them by making the acquisition of needed low-level details as simple as asking a question. Where, before, they gathered clues that might suggest to them a fault's immediate cause by searching traces and single-stepping through a program run, now they can simply ask questions to gather such clues. The programmers still direct the debugging process, but the question-answering system helps them to determine the next step in that process. By allowing programmers to spend less time single-stepping through program runs and searching traces for historical details, this question-answering system can let programmers save their attention and energy for the high-level aspects and intuition needed in debugging.

As already mentioned, CLIPS 5.0 forms the basis of MIRO: the CLIPS 5.0 source code was modified and augmented to implement MIRO. To quickly obtain an operational prototype, we used existing code whenever possible, and we patterned the historical Rete and agenda reconstruction additions after those used for regular Rete within CLIPS. This software reuse included replicating the code for existing data structures when creating new data structures, augmenting existing data structures, calling existing functions from new functions whenever possible, and modifying existing functions when their functionality was almost, but not quite, what was needed.

So, when implementing, for example, historical Rete's past partitions, the past partitions were patterned after the current (formerly only) partitions. The data structures for facts, instantiations, and rule instantiations were all augmented with time-tags, and rule instantiations were also augmented with a fired flag, set if that rule instantiation was actually fired. We added analogous functions to those used for maintaining the current working memory for maintaining the past working memory, and so on. This approach reduced programming overhead, and, because CLIPS 5.0 was already quite efficient, the added historical components were also quite efficient.

We also added code to measure various run-time characteristics, such as the number of current and past facts, rule instantiations, and Rete memory instantiations, to better compare program runs under regular CLIPS and MIRO, as shown at the end of section four.

The bulk of MIRO was implemented over the course of a single year, by a single programmer; however, other research-related activities were being done concurrently with this development. It

probably took about seven programmer-months to bring MIRO to the point where it had 19 question-types. Note, however, that this does not take into account the time spent designing the historical Rete and agenda reconstruction algorithms.

The version of CLIPS 5.0 that we developed MIRO from, and that we used for comparisons with MIRO, contained 80497 lines of code and comments; this includes the source code files, include files, and the UNIX make file. The same files for MIRO contained 86099 lines, also including comments; so, we added about 5600 additional lines of code and comments, making MIRO about 7% larger than CLIPS 5.0.

3. MIRO'S QUESTION TYPES

Adding the question-answering system itself to MIRO was as easy as adding a new user-defined function to the top-level of CLIPS; we constructed a CLIPS top-level function called askquestion. The difficult part was determining the form that this question-answering would take. Our primary goals of constructing a prototype both to demonstrate the feasibility of, and to illustrate how, the information from the historical Rete network and other history-related structures could be used to answer programmers' debugging-related questions had a strong impact on the design that we decided to use.

We assume that the kinds of questions that can be asked are limited to instances of a set of fixed question-types; each question-type can be thought of as a template for a particular kind of question. This allows the question-answering system to have a modular design: each question-type has an algorithm for answering an instance of that type. This also allows additional question-types to be easily added, and to be tested as they are added, as it is discovered which are desirable for debugging.

The design of the interface for getting programmers' questions to the explanation system is a worthy research topic all by itself; to allow us to concentrate more on answering the questions, we use a very simple interface, with the understanding that future work could include replacing this simple interface with a more sophisticated front-end. Since we are more interested in designing a tool for debugging and less interested in natural language processing, the question-answering system uses a template-like approach for the various question-types that the programmers will be able to ask. That is, each question-type has a specific format, with specific ``blanks" which, filled in, result in an instance of that question-type. Furthermore, to avoid requiring the programmers to memorize these question-type formats, we use a menu-based approach: when the programmers enter the command (askquestion), a list of currently-supported question-types is printed on-screen. They then enter the number of the desired question-type, and the explanation system queries them to fill in that question-type's necessary blanks.

Implementing this approach was quite straightforward, because regular CLIPS already has some tools for obtaining top-level command arguments. We only had to modify them a little to allow for the optional askquestion arguments. A main askquestion function prints the menu if no arguments are given, and then asks which question-type is desired; a case statement then uses either that result or the first askquestion argument to call the appropriate question-answering function, which is in charge of seeing if arguments for its needed values have already been given, or must be asked for. After each particular question-type's question-answering function obtains or verifies its needed values, it then tries to answer the question, and print the result for the programmer.

We currently support 19 question-types, as shown in Figure 2. However, question 9, why a particular rule did not fire, currently only tells if that rule was eligible or not at the specified time.

1. What fact corresponds to fact-id <num>?
2. When did rule <name> fire?
3. What rule fired at time <num>?
4. What facts were in memory at time <num>?
5. How many current facts are in memory now?
6. How many past facts are in memory now?
7. How many current rule activations are on the agenda now?
8. How many past rule activations are in memory now?

```
9.  Why did rule <name> not fire at time <num>?
10. How many current alpha instantiations are in memory now?
11. How many past alpha instantiations are in memory now?
12. How many current beta instantiations are in memory now?
13. How many past beta instantiations are in memory now?
14. What are the Rete memory maximums?
15. What were the agenda changes from time <num> to time <num>?
16. How many current not-node instantiations are in memory now?
17. How many past not-node instantiations are in memory now?
18. What was the agenda at the end of time <num>?
19. How many agenda changes were there from time <num> to time <num>?
```

Figure 2 - Currently-Supported Question-Types in MIRO

We will now give some examples of MIRO's question-answering system at work. We will describe some hypothetical scenarios, to illustrate how MIRO might be useful in debugging; the responses shown are those that would be given by MIRO in such situations.

Consider a program run in which the program ends prematurely, that is, without printing any output. One can find out the current time-counter value with a command that we added to MIRO specifically for this purpose --- if one types (time-counter) after 537 rule-firings, it prints out:

```
time_counter is: 537
```

The programmers can now ask, if desired, which rules fired at times 537, 536, 535, etc. If they type (askquestion), the menu of rules will be printed; if they choose question-type number 3, ``What rule fired at time <num>?", then it will ask what time-counter value they are interested in; if 537 is entered, and if a rule named ``tryit" happened to be the one that fired at that time, then MIRO would print an answer like:

```
Rule tryit fired at time 537
```

with the following rule activation:

```
0     tryit: f-30,f-15,f-47 time-tag: (530 *)(activn time-tag: 530 537))
```

This tells the programmers that rule tryit fired at time 537, and that the particular instantiation of rule tryit that fired had LHS conditions matched by the facts with fact-identifiers f-30, f-15, and f-47. This instantiation of rule tryit has been eligible to fire since time 530 --- before the 531st rule firing --- but, as shown, the rule instantiation's, or activation's, time-tag is now closed, with the time 537, because it was fired then, and a rule that is fired is not allowed to fire again with the same fact-identifiers.

Now, if the programmers suspect that this rule-firing did not perform some action that it should have performed --- to allow another rule to become eligible, for example --- then they can use the regular CLIPS command ``pprule" to print the rule, so that they can examine its RHS actions. If it should not have fired at all, then they may wish to see why it was eligible. For example, in this case, they may want to know what facts correspond to the fact-identifiers f-30, f-15, and f-47. One can look at the entire list of fact-identifiers and corresponding facts using the regular CLIPS (facts) command, but if the program has very many facts, it can be quite inconvenient to scroll through all of them. So, MIRO provides the question-type ``What fact corresponds to fact-id <num>?". On first glance, this question appears to have no historical aspect at all; however, it does include the time-tag for the instance of the fact corresponding to this fact-identifier. This can be helpful to the programmers, if they suspect that one of the facts should not have been in working memory --- then, the opening time of that fact's time-tag can be used to see what rule fired at that time, probably resulting in this fact's addition. Since this question-type is the first in the list, and requires as its only additional information the number of the fact identifier whose fact is desired, typing (askquestion 1 47) will ask the first question for fact-identifier f-47, giving an answer such as:

```
Fact-id 47 is fact:
   (p X Y) time-tag: (530 *)
```

If the programmers suspect that fact f-47, now known to be (p X Y), should not be in working

memory --- if they think that it is a fault that it exists, and is helping rule tryit to be able to fire --- then they can again ask the question-type "What rule fired at time <num>?" to see what rule fired at time 530, when this instance of (p X Y) joined working memory. They can then see if the rule that fired at time 530 holds the cause of the fault of (p X Y) being in working memory, and enabling the faulty firing of tryit at time 537.

4. COMPARISONS BETWEEN MIRO AND CLIPS 5.0

As already mentioned, we implemented MIRO by starting with CLIPS 5.0; we then generalized its Rete inference network [2] into a *historical Rete* network, added an agenda reconstruction capability, and added the prototype question-answering capability. Historical Rete and agenda reconstruction are discussed in more detail in [5] and [6]. We also made some other modifications, to allow for experimental measures; for example, we added code to measure various run-time characteristics such as the number of current and past instantiations, and the number of current and past facts. We then ran a number of programs under both MIRO and CLIPS 5.0.

The programs that we used range fairly widely in size, and behavior. Four of the programs --- dilemma1, mab, wordgame, and zebra --- are from CLIPS 5.0's Examples directory. The dilemma1 program solves the classic problem of getting a farmer, fox, goat, and cabbage across a stream, where various constraints must be met. The mab program solves a planning problem in which a monkey's goal is to eat bananas. The wordgame program solves a puzzle in which two six-letter names are "added" to get another six-letter name, and the program determines which digits correspond to the names' letters. Finally, the zebra program solves one of those puzzles in which five houses, of five different colors, with five different pets, etc., are to each have their specific attributes determined, given a set of known information.

The AHOC program was written by graduate students in the University of Houston Department of Computer Science's graduate level knowledge engineering course COSC 6366, taught by Dr. Eick in Spring 1992. AHOC is a card game with the slightly-different objective that the players seek to win *exactly* the number of tricks bid. The program weaver ([1], [3]) is a VLSI channel and box router; we obtained a CLIPS version of this program from the University of Texas' OPS5 benchmark suite. Finally, can_ordering_1 is a small program that runs a rather long canned beverage warehouse ordering simulation, also from the Spring 1992 COSC 6366 knowledge engineering class; it was written by C. K. Mao.

We ran each program three times under MIRO and under CLIPS 5.0, on a Sun 3 running UNIX, with either no one else logged in, or one other user who was apparently idle. For every run in both CLIPS and MIRO, we used the (watch statistics) command to check that the same number of rules fired for each run of the program; for every MIRO run of a program, we also made sure that all runs had the same number of instantiations at the end of the run. The run-times are given in Table I.

The run-times for programs run using MIRO were usually only slightly slower than those using regular CLIPS 5.0; one program, mab, took 11.4% more time in MIRO, but on average, the MIRO runs only took 4.1% more time. Interestingly enough, AHOC ran, on average, slightly faster under MIRO than under regular CLIPS. This could be because regular CLIPS 5.0 returns the memory used for facts and instantiations to available memory as they are removed, which MIRO does not do until a reset or clear, because such facts and instantiations are instead kept, and moved into the past fact list or past partitions. The average 4.1% additional time required by MIRO to run a program seems quite reasonable, especially since the additional time is only required while debugging, and allows programmers the benefits of the MIRO tool.

Another feature of the (watch statistics) command under regular CLIPS, besides printing the number of rules fired and the time elapsed, is that it also prints the average and maximum number of facts and rule instantiations during those rule-firings. We enhanced this command in MIRO so that it also keeps track of the average and maximum number of past facts and past rule instantiations, as well as the averages and maximums for different kinds of Rete memory instantiations, both past and current. To get some idea of the overhead needed to store historical information from a run, we compared the average number of current facts during a run to the average number of past facts, and compared the

Table I
Run-Times

program	which CLIPS	# rules fired	run times (in sec.) run1	run2	run3	avg run-time	% slower
dilemma1	regular	80	0.80	0.88	0.80	0.83	
	MIRO	80	0.88	0.84	0.94	0.89	7.3%
mab	regular	81	2.02	1.94	2.06	2.01	
	MIRO	81	2.26	2.18	2.28	2.24	11.4%
wordgame	regular	102	16.84	16.50	16.58	16.64	
	MIRO	102	16.78	16.74	16.62	16.71	0.4%
zebra	regular	28	7.70	7.02	6.96	7.23	
	MIRO	28	7.88	7.16	7.20	7.41	2.5%
AHOC	regular	2747	60.34	60.28	60.52	60.38	
	MIRO	2747	59.10	58.62	60.28	59.33	-1.8%
weaver	regular	745	198.26	198.94	201.24	199.48	
	MIRO	745	209.10	208.84	207.94	208.63	4.6%
can_ordering_1	regular	3392	174.20	174.26	179.20	175.89	
	MIRO	3392	186.18	181.34	181.58	183.03	2.1%

average number of current rule instantiations to the average number of past rule instantiations.

Another feature of the (watch statistics) command under regular CLIPS, besides printing the number of rules fired and the time elapsed, is that it also prints the average and maximum number of facts and rule instantiations during those rule-firings. We enhanced this command in MIRO so that it also keeps track of the average and maximum number of past facts and past rule instantiations, as well as the averages and maximums for different kinds of Rete memory instantiations, both past and current. To get some idea of the overhead needed to store historical information from a run, we compared the average number of current facts during a run to the average number of past facts, and compared the average number of current rule instantiations to the average number of past rule instantiations. (Analogous comparisons for different kinds of Rete memory instantiations, as well as comparisons of the maximum number of current items to the maximum number of past items, can be found in [6]).)

Table II shows these averages for facts and rule instantiations. It also gives, where appropriate, how many times bigger the average number of past items is than the average number of current items. Compared to the average number of current items during a run, fewer times as many past facts than past rule instantiations will need to be kept. This suggests that there is more overhead to storing rule instantiation history than there is to storing fact history.

Table II shows that the space to store past facts and rule instantiations, compared to the average space to store current facts and rule instantiations, can, indeed, be high, especially for long runs. But, long runs should be expected to have more historical information to record. Also, it should be noted that in running these programs on a Sun 3, we never ran into any problems with space, even with the additional historical information being stored. During program development, the storage costs should be acceptable, since they facilitate debugging-related question-answering. The programmers can also limit the storage costs by only running their program using MIRO when they might want to obtain the answers to debugging-related questions; at other times, they can run their program using regular CLIPS.

5. USING MIRO IN STUDYING CLIPS PROGRAMS

We hope that MIRO's question-answering system can make debugging a CLIPS program easier and less tedious. Here, we consider another use of MIRO: to analyze certain performance aspects of CLIPS programs. With a shift of viewpoint, such analysis may be involved in a variant of debugging -

Table II
Average No. of Facts and Rule Instantiations

program	avg. # of facts		times	avg. # rule insts		times
	current	past	bigger	current	past	bigger
dilemma1	11	20	1.82	6	72	12.00
mab	19	42	2.21	1	64	64.00
wordgame	71	0		49	51	1.04
zebra	78	14	0.18	11	14	1.27
AHOC	231	792	3.43	107	3447	32.21
weaver	151	383	2.54	6	689	114.83
can_ordering_1	113	1639	14.50	10	2795	279.50

-- if, for example, a program takes so long to run that the programmers consider the run-time to be a problem, then such performance analysis may help in determining how to modify the program so that it takes less time. Such aspects may also be of interest in and of themselves, both to programmers and to researchers studying the performance aspects of forward-chaining programs in general.

Note that a Rete network, historical or regular, encodes a program's rules' LHS conditions; the RHS conditions of those rules are not represented, except perhaps by pointers from a rule's final beta memory to its actions, to help the forward-chaining system to more conveniently execute the RHS actions of a fired rule. So, MIRO could be helpful primarily in analyzing the efficiency involved in the match step of the recognize-act cycle. Being able to locate Rete memories whose updating could cause performance trouble-spots could be useful for improving the overall performance of a CLIPS program.

Using MIRO, it is much easier to discover some of the dynamic features of a CLIPS program run, such as the number of instantiations within the network during a run. One can study worst-case and average-case behavior within a CLIPS program run by looking at the number of average, and maximum, facts, rule instantiations, and alpha, beta, and not-memory instantiations. For example, a great disparity between the average number of current beta instantiations, and the maximum number reached during a run could indicate volatility in beta memory contents that could have a noticeable performance impact.

One might consider the total number of changes to a beta memory during a run to be the total number of additions to and deletions from that memory --- or, the total number of current instantiations at the end plus two times the number of past instantiations. Averaged over the number of rule-firings, this would give the number of beta memory changes per rule-firing, which, if high, might very well correlate with more time needed per rule-firing; and, averaged over the number of total beta memories, this would give us a rough average of the number of changes per beta memory. We could even determine a number of working memory changes this way, by adding the number of current facts to two times the number of past facts, and use this to obtain an average number of fact changes per rule firing.

Another measure that might be very telling would be the average number of memory changes per rule-firing, computed by counting each past instantiation at the end of a run as two memory changes --- since each was first added to a current instantiation, and then moved to a past partition --- and each current instantiation at the end of a run as a single memory change, and then dividing the sum by the number of rule-firings. We can also determine the average number of fact changes per rule-firing similarly. In measurements we made using the seven programs discussed in the previous section, the most important factor that we found that correlated with performance was that a high number of instantiation changes per rule-firing did seem to correlate with more time needed per rule-firing [6].

Although one can reasonably obtain some of the information mentioned above by using regular CLIPS, much of it would be very inconvenient to gather using it. For example, determining the

average and maximum number of past facts and past rule instantiations would be difficult to obtain using regular Rete. We modified the existing CLIPS (watch statistics) command in MIRO so that it also keeps track of this additional information.

Note that historical Rete can also be used to support trouble-shooting tools and question-types, in addition to supporting question-answering for debugging. For example, any time that the programmer can specify a particular time of interest, historical Rete searches can be made that focus only on instantiations in effect at that time.

The discussed examples show the potential that MIRO has as a tool in analyzing CLIPS program performance, as well as in debugging. Information about what occurred during Rete network memories can be more reasonably retrieved, making such analysis more practical, across larger samples of programs. Note, too, that a programmer can choose to look at averages over all of a program's memories, or for a single memory, or for a particular rule's memories, as desired.

6. CONCLUSIONS

In this paper, we have followed up on our work reported at the last CLIPS conference, describing how we have implemented historical Rete and question-answering for debugging in MIRO, a debugging tool built on top of CLIPS 5.0. We have described MIRO, and have hopefully given a flavor for how it may be used to make debugging a CLIPS program easier and less tedious, by allowing programmers to simply ask questions to determine when program events --- such as rule firings, or fact additions and deletions --- occurred, instead of having to depend on program traces or single-stepping program runs. We have further described how MIRO might be used to study certain performance-related aspects of CLIPS programs.

The empirical measures included also show that MIRO's costs are not unreasonable. Comparisons of programs run in both MIRO and CLIPS 5.0, which MIRO was built from, have been given; on average, the run-time for a program under MIRO was only 4.1% slower than a program run under CLIPS 5.0, when both were run on a Sun 3. Comparing the average number of past facts and rule instantiations to the average number of current facts and current rule instantiations, there were, on average, 3.53 times more past facts than current facts, and 72.12 times more past rule instantiations than current rule instantiations. But this historical information permits the answering of debugging-related questions about what occurred, and when, during an expert system run. We also gave examples of the kinds of question-types that MIRO can currently answer, as well as examples of the kinds of answers that it gives. We hope that this research will encourage others to also look into how question-answering systems can be designed to serve as tools in the development of CLIPS programs.

REFERENCES

1. Brant, D. A., Grose, T., Lofaso, B., Miranker, D. P., "Effects of Database Size on Rule System Performance: Five Case Studies," Proceedings of the 17th International Conference on Very Large Data Bases (VLDB), 1991.

2. Forgy, C. L., "Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem," ARTIFICIAL INTELLIGENCE, Vol. 19, September, 1982, pp. 17-37.

3. Joobbani, R., Siewiorek, D. P., "WEAVER: A Knowledge-Based Routing Expert," IEEE DESIGN AND TEST OF COMPUTERS, February, 1986, pp. 12-23.

4. Tuttle, S. M., Eick, C. F., "Adding Run History to CLIPS," 2nd CLIPS Conference Proceedings, Vol. 2, Houston, Texas, September 23-25, 1991, pp. 237-252.

5. Tuttle, S. M., Eick, C. F., "Historical Rete Networks to Support Forward-Chaining Rule-Based Program Debugging," INTERNATIONAL JOURNAL ON ARTIFICIAL INTELLIGENCE TOOLS, Vol. 2, No. 1, January, 1993, pp. 47-70.

6. Tuttle, S. M., "Use of Historical Inference Networks in Debugging Forward-Chaining Rule-Based Systems," Ph.D. Dissertation, Department of Computer Science, University of Houston, Houston, Texas, December, 1992.

Optimal Pattern Distributions in Rete-based Production Systems

Stephen L. Scott

Hughes Information Technology Corporation
1768 Business Center Drive, 4th Floor
Reston, VA 22090
(703) 759-1356

sscott@mitchell.hitc.com

ABSTRACT

Since its introduction into the AI community in the early 1980's, the Rete algorithm has been widely used. This algorithm has formed the basis for many AI tools, including NASA's CLIPS. One drawback of Rete-based implementations, however, is that the network structures used internally by the Rete algorithm make it sensitive to the arrangement of individual patterns within rules. Thus while rules may be more or less arbitrarily placed within source files, the distribution of individual patterns within these rules can significantly affect the overall system performance. Some heuristics have been proposed to optimize pattern placement, however, these suggestions can be conflicting.

This paper describes a systematic effort to measure the effect of pattern distribution on production system performance. An overview of the Rete algorithm is presented to provide context. A description of the methods used to explore the pattern ordering problem area are presented, using internal production system metrics such as the number of partial matches, and coarse-grained operating system data such as memory usage and time. The results of this study should be of interest to those developing and optimizing software for Rete-based production systems.

INTRODUCTION

The Rete algorithm was developed by Charles Forgy at Carnegie Mellon University in the late 1970's, and is described in detail in [Forgy, 1982]. Rete has been used widely in the expert system community throughout the 1980's and 1990's, and has formed the basis for several commercial and R&D expert system tools [Giarratano & Riley, 1989] [ILOG, 1993]. Recent enhancements have been proposed based on parallel processing [Miranker, 90] and matching enhancements [Lee and Schor, 1992]. Rete provides an efficient mechanism for solving the problem of matching a group of facts with a group of rules, a basic problem in a production system.

In this section, an overview of the Rete algorithm is given in order to provide context for the discussion to follow. This presentation, however, is not intended to be a rigorous analysis of the Rete algorithm.

Rete based systems assume a working memory that contains a set of facts and a network of data structures that have been compiled from rule definitions. The rules contain a set of condition elements (CE's) that form the left-hand-side (LHS), and a right-hand-side

(RHS) that performs actions. The RHS actions may be side-effect free, such as performing a computation, invoking an external routine, performing I/O to the input or output streams or file. Other actions on the RHS may cause changes in the working memory, such as insertions, deletions, or modifications of facts. The Rete network actually contains two main structures: a pattern network, and a join network. The pattern network functions to identify which facts in working memory are associated with which patterns in the rules. The join network is used to identify which variables are similarly bound within a rule across CE's.

Within the pattern network, elements of the individual CE's are arranged along branches of a tree, terminating in a leaf node that is called an alpha node. The join network consists of groups of beta nodes, each containing two nodes as inputs and one output that can be fed to subsequent beta nodes. Finally, the output of the join network may indicate that one or more rules may be candidates for firing. Such rules are called activations, and constitute inputs to the conflict set, which is a list of available rules that are ready for execution. Typically, some arbitration mechanism is used to decide which rules of equal precedence are fired first. When a rule fires, it may of course add elements to or delete elements from the working memory. Such actions will repeat the processing cycle described above, until no more rules are available to be fired.

Consider the following small set of facts and a rule. For simplicity, additional logical constructs, such as the TEST, OR, or NOT expressions are not considered, and it is assumed that all CE's are ANDed together, as is the default. Note that myRule1 has no RHS, as we are focusing only on the LHS elements of the rule.

```
(deffacts      data
       (Group        1 2 3)
       (Int 1)
       (Int 2)
       (Int 3))

(defrule       myRule1
       (Group        ?i ?j ?k)
       (Int ?i)
       (Int ?j)
       (Int ?k)
=>)
```

This rule can be conceptualized in a Rete network as follows (see Figure 1). There are two branches in the pattern network, corresponding to the facts that begin with the tokens "Group" and "Int", respectively. Along the "Group" branch of the tree, there are nodes for each of the tokens in the fact, terminating with an alpha node that contains the identifier "f-1" corresponding to the first fact in the deffacts data defined above. Similarly, along the "Int" branch, there is one node for all the facts that have "Int" as a first token, and then additional nodes to show the various values for the second token. Alpha nodes along this branch also contain references to the appropriate facts that they are associated with, numbered in the diagram as "f-2" through "f-4". Note that the "Int" branch has shared nodes for structurally similar facts, i.e. there is only on "Int" node even though there are three facts with "Int" as a first token.

On the join network, myRule1 has three joins to consider. The first CE of myRule1 requires a fact consisting of a first token equal to the constant "Group" followed by three additional tokens. The alpha node of the "Group" branch of the pattern network supplies one such fact, f-1. The second CE of myRule1 requires a fact consisting of a first token

equal to the constant "Int" followed by another token, subject to the constraint that this token must be the same as the second token of the fact satisfying the first CE. In this case, the fact f-2 meets these criteria, hence the join node J1 has one partial activation. This is because there is one set of facts in the working memory that satisfy its constraints. Continuing in this fashion, the output of J1 is supplied as input to J2, which requires a satisfied join node as a left input and a fact of the form "Int" followed by a token (subject to the constraint that this token must be equal to the third token of the first CE). The fact f-3 meets these criteria, so join node J2 has one partial activation as well. This process continues until we finish examining all CE's in myRule1 and determine that there are indeed facts to satisfy the rule. The rule is then output from the join network with the set of facts that satisfied its constraints and sent on to the agenda, where it is queued up for execution.

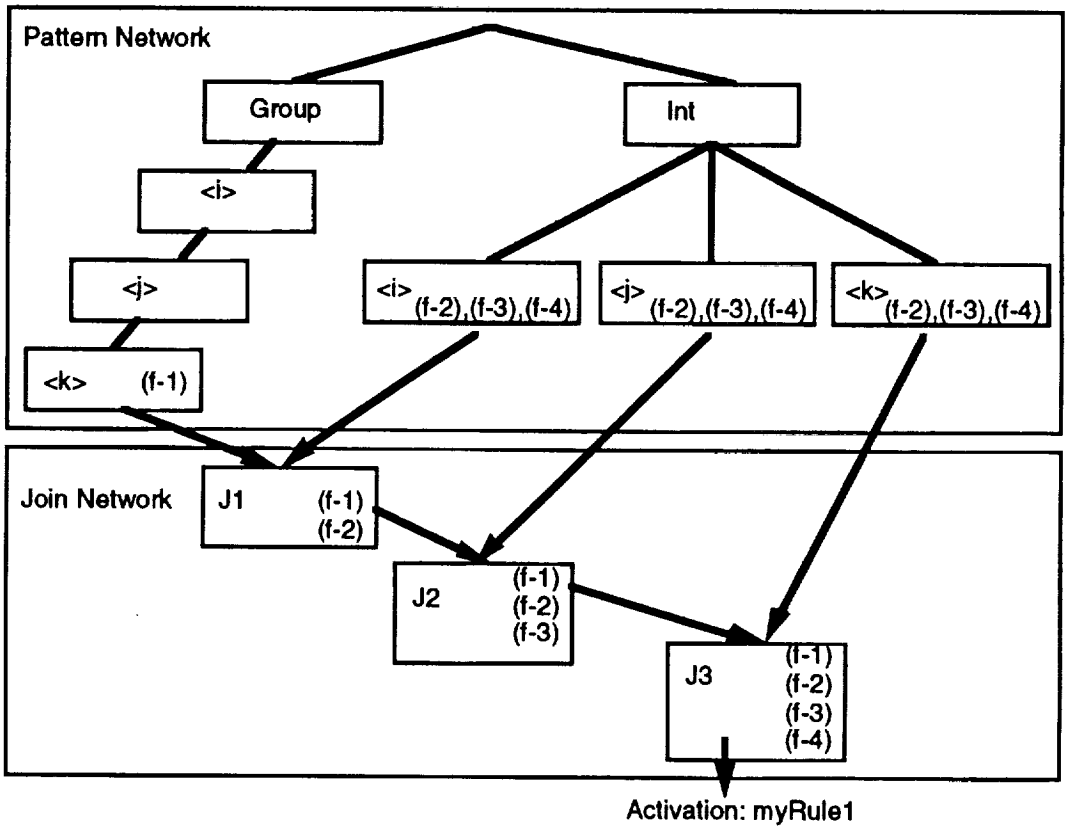


Figure 1. Rete Network for myRule1. This diagram depicts the Rete pattern and join networks for a rule with four CE's.

Consider the same set of data and a rule with the CE's arranged in a different order. Semantically, myRule1 and myRule2 are the same, however, the number of partial matches generated by myRule2 is much greater than that generated by myRule1.

```
(defrule      myRule2
      (Int ?i)
      (Int ?j)
      (Int ?k)
      (Group      ?i ?j ?k)
=>)
```

With this rule, we have three facts that match the first CE. Examining the first two CE's, there are three possible facts that can match the second CE, hence there are nine possible ways to satisfy the first two CE's. Moving to the third CE, there are again three ways to satisfy the third CE, but each of these must be considered with the nine possibilities that preceded it, hence there are 27 possible ways to satisfy the first three CE's. Fortunately, the fourth CE is satisfied by only one fact, so the number of partial activations for CE's one through four is only one; it is the fact that matches the fourth CE (f-1), coupled with exactly one set of the 27 possibilities available for CE's one through three. Summarizing, there are 40 partial activations for this rule (3 + 9 + 27 + 1).

From the above discussion, we have seen that pattern ordering in the LHS of rules can have significant impact on performance. Unfortunately, there are a large number of possible orderings one can try in even a small rule. Since in general, in a rule with N CE's, there are N ways to place the first CE, N-1 ways to place the second CE, and so on, the number of possible pattern arrangements is given by N!. As there may be many rules in an expert system, each with a large number of possible CE orderings, it should be clear that it is prohibitively expensive to do an exhaustive search of all possible arrangements of all rules in an attempt to optimize performance.

There may be some reduction in the number of arrangements if one considers that from the pattern network point of view, some arrangements produce an equal number of partial activations and thus can be considered together for analysis purposes. For example, the rule aRule1

```
(defrule aRule1
        (Group ?i ?j)
        (Int ?i)
        (Int ?j)
=> )
```

has the same number of partial activations as aRule2,

```
(defrule aRule2
        (Group ?i ?j)
        (Int ?j)
        (Int ?i)
=> )
```

because the CE's in slots 2 and 3 are similar with respect to effect on the join network. So even though there are 6 possible arrangements of CE's in this rule, only 3 actually produce different numbers of partial activations in the join network. This property is used extensively in the Analysis section that follows, as it allows valid results to be obtained using a manageable subset of the possible pattern orderings.

ANALYSIS AND RESULTS

In an effort to better understand the effects of pattern ordering on production system performance, a series of tests were conducted. This section describes the various experiments and the results obtained.

Partial Activations

From the discussion above, it is evident that some CE orderings are considerably more efficient than others with respect to partial activations. A test suite was developed using a rule with N CE's and a data set of N facts, where N-1 CE's are syntactically similar and one CE joins across the remaining N-1 CE's. This is the configuration used in the example myRule1 and myRule2 above; note that in that case N=4 because there are four CE's.

To interpret the data in the following table, match the number of CE's in the rule LHS (identified by the row labeled N = <n>) with the position of the constraining fact (the fact that has elements to match all other CE's in the rule). For the example myRule1 cited above, the row "N=4" is matched with "Fact Pos 1", giving 4 partial activations. Similarly, the example myRule2 cited above has a constraining fact in position 4, hence for N=4, the number of partial activations is 40. The following table shows the results of the number of partial activations for rules with the number of CE's varying from N=2 to N=8.

	Fact Pos 1	Fact Pos 2	Fact Pos 3	Fact Pos 4	Fact Pos 5	Fact Pos 6	Fact Pos 7	Fact Pos 8
N = 2	2	2						
N = 3	3	4	7					
N = 4	4	6	14	40				
N = 5	5	8	23	86	341			
N = 6	6	10	34	158	782	3906		
N = 7	7	12	47	262	1557	9332	55987	
N = 8	8	14	62	404	2804	19610	137258	960800

Table 1. Partial Activations in Rule Sets. This table shows the increase in partial activations observed in rules with various numbers of CE's, where a constraining fact is located at the position indicated by the column heading.

From this, at least two observations may be made. First, it is clear that the number of partial activations grows very rapidly. For this example set of rules and data, the number of partial activations for a rule with N CE's is given by

$$\sum_{i=0}^{N-1} (N-1)^i \qquad (1.0)$$

With such growth, on small computer systems, this may result in unexpected termination of a program, and even on large systems, performance may be degraded as the system attempts to accommodate the memory demands through paging or swapping. The second observation is the smaller the number of CE's on the LHS, the smaller the upper limit on partial activations. This suggests that a system with a larger number of smaller rules is better, at least from the vantage point of partial activations, than a system with a smaller number of larger rules.

Memory Usage

Within the Rete network implementation, data is maintained about partial activations. This data requires memory allocation, and as expected, the required memory grows in proportion to the number of partial activations. To examine this, the same suite of rules used above for partial activation testing was used, however, in this case, calls were made to the CLIPS internal function (mem-used) in order to calculate the memory required to store a network. The following table shows the results of these tests.

	Fact Pos 1	Fact Pos 2	Fact Pos 3	Fact Pos 4	Fact Pos 5	Fact Pos 6	Fact Pos 7	Fact Pos 8
N = 2	376	376						
N = 3	548	560	560					
N = 4	596	620	620	960				
N = 5	836	872	872	1912	8008			
N = 6	960	1008	1008	3228	18180	105652		
N = 7	1016	1076	1076	5076	36132	253832	1746792	
N = 8	1292	1364	1364	7864	65440	536008	4300744	33947716

Table 2. Memory Requirements for Various Rule Sets. This table shows the increase in memory requirements observed in rules with various numbers of CE's, where a constraining fact is located at the position indicated by the column heading. Memory allocation values are in bytes.

As expected, the amount of memory required to represent a rule varies in proportion to the number of partial activations. The two observations given for partial activations also hold here: some rule LHS orderings will require much less memory than others, and it is in general more memory efficient to have more small rules than a few large rules.

Reset Time

After rules and data are read into the system, the network must be updated to reflect the state required to represent these constructs. Data must be filtered through the network in order to determine facts are available, and comparisons must be made across CE's to determine which rules are eligible for firing. In order to investigate the time these processes take, the same test suite describe above was used, however, in this case, an operating system call was used to time the execution of the load and reset operations for the various rules. The "timex" command, available on many systems, gives operating system statistics about the real time, system time and user time required to execute a process. The following table shows the results of this test, giving real time in seconds, for the test suite.

	Fact Pos 1	Fact Pos 2	Fact Pos 3	Fact Pos 4	Fact Pos 5	Fact Pos 6	Fact Pos 7	Fact Pos 8
N = 2	0.1	0.1						
N = 3	0.1	0.1	0.1					
N = 4	0.1	0.1	0.1	0.1				
N = 5	0.1	0.1	0.1	0.1	0.11			
N = 6	0.1	0.11	0.11	0.11	0.11	0.17		
N = 7	0.1	0.1	0.13	0.11	0.12	0.25	0.96	
N = 8	0.11	0.1	0.11	0.11	0.15	0.45	2.51	17.88

Table 3. Reset Time for Rule Sets. This table shows the increase in reset time observed in rules with various numbers of CE's, where a constraining fact is located at the position indicated by the column heading.

As the reset times do not grow as rapidly as N increases, these results suggest that reset time is not as great a consideration as memory or number of partial activations. Also the granularity of timex is only 1/100 of a second, making more precise measurements difficult.

Placement of Volatile Facts

One heuristic that has been proposed concerns the placement of volatile facts in a rule. In data sets where a particular type of pattern is frequently asserted or retracted (or modified if the tool supports this), it is best to put these patterns at the bottom of the LHS of the rule. A typical example is a control fact containing constants, typically used to govern processing phases. The justification given is that because Rete attempts to maintain the state of the system across processing cycles, by placing the volatile fact at the bottom of the LHS, Rete does not need to check most of the rest of the network and can realize some performance gain. To test this, the following scenario was used. The data set consisted of a set of facts of the form

```
(Int    val <n>        isPrime Yes)
```

where <n> contained a prime number in the range 1 <= n <= 1000. A volatile counter fact of the form

```
(Counter <n>)
```

was used, where n again ranged from 1 <= n <= 1000. This fact was asserted and retracted for each value of n in the range. The rules to test whether or not n was prime were

```
(defrule        isPrime
        (Int    val ?n  isPrime Yes)
        ?x <- (Counter ?n)
=>
        (retract ?x)
        (assert (Counter =(+ ?n 1))
        (printout t ?n " is a prime " crlf))
```

```
(defrule        notPrime
        (Int    val ?n  isPrime Yes)
        ?x <- (Counter ?ctrVal&:(!= ?n ?ctrVal))
=>
        (retract ?x)
        (assert (Counter =(+ ?ctrVal 1))
        (printout t ?ctrVal " is not a prime " crlf))
```

The results below indicate run times in seconds for systems that searched for primes up to size K. The column 100, for example, indicates that primes between 1 and 100 were sought by using the volatile fact (Counter <n>) 100 times.

The example rules isPrime and notPrime given above correspond the rules used for the "volatile fact at bottom" row of the table. The "volatile fact at top" rules are virtually the same, except that the (Counter <n>) fact appears as the first CE instead of the second as illustrated above.

	100	250	500	750	1000
volatile fact at top	1.67	4.74	8.17	15.01	20.31
volatile fact at bottom	1.39	3.92	8.06	12.19	16.43

Table 4. Run Times for Rules with Volatile Facts. This table shows the differences in run times observed in rules with volatile facts placed at the top or bottom of the LHS. Times are in seconds.

This example shows that placing volatile facts at the bottom of a rule improves runtime performance, even for a small rule set and small amounts of data. The improvement is more obvious as the problem size grows, as the observed difference for K=100 is slight, whereas the difference for K=1000 is almost 4 seconds.

Placement of Uncommon Facts

Another heuristic suggests that facts that are relatively rare in the system should be placed first on the LHS. To test this, the following scenario was used. A data set contained three classes of facts: sensor facts, unit facts, and controller facts. These facts were distributed in the system in various proportions. Two rules were compared, one organized so that its CE's matched the distribution of the facts, and the other exactly opposite. In the following rules, rareFirst is tailored to perform well when the number of Ctrl facts is less than the number of Unit facts and the number of Unit facts is less than the number of Sensor facts. Conversely, rareLast is not expected to perform as well under this arrangement of data.

```
(defrule        rareFirst
        (Ctrl           Id ?cid         Status ?cStat)
        (Unit           Id ?uid         Ctrl ?cid       Status ?ustat   Value ?uVal)
        (Sensor         Id ?sid         Unit ?uid       Value ?sVal)
=>)
```

```
(defrule      rareLast
      (Sensor      Id ?sid      Unit ?uid      Value ?sVal)
      (Unit        Id ?uid      Ctrl ?cid      Status ?ustat   Value ?uVal)
      (Ctrl        Id ?cid      Status ?cStat)
=>)
```

The following table shows the number of partial activations generated for these rules given various distributions of matching Ctrl, Unit, and Sensor facts. The nomenclature i:j:k indicates that there were i Ctrl facts, j Unit facts, and k Sensor facts.

	Ctrl:Unit: Sensor 3:10:20	Ctrl:Unit: Sensor 5:20:50	Ctrl:Unit: Sensor 10:50:100	Ctrl:Unit: Sensor 25:125:500	Ctrl:Unit: Sensor 50:200:1000
rarest fact at top	33	75	160	650	1250
rarest fact at bottom	60	150	300	1500	3000

Table 5. Partial Activations for Rules with Rare Facts. This table shows the differences in partial activations observed in rules with patterns that match rarest facts at the top or bottom of the LHS.

This test shows that placing less common facts at the top of the LHS reduces the number of partial activations for the rule. Another point is worthy of mention here: had the distribution of facts been different, rareLast might have outperformed rareFirst rule. This points out a potential problem, as attempting to optimize a system based on one set of data may not have optimal results on other sets of data. Given that expert systems are typically much more data driven than other forms of software, this kind of optimization may not be effective if the data sets vary widely.

CONCLUSIONS

This paper has described a number of tests performed to investigate the effects of pattern ordering on production system performance. The results have borne out widely held heuristics regarding pattern placement on the LHS of rules. The results have quantified various aspects of the problem of partial activation growth by measuring the number of partial activations, memory requirements, system reset and run time for a variety of pattern configurations.

In general, the conclusions that can be drawn are as follows. Partial activations can vary exponentially as a result of pattern ordering. This suggests that (1) rules should be written with some regard to minimizing partial activations, and (2) systems should use larger numbers of small rules rather than smaller numbers of large rules. The second suggestion helps to reduce the risk of having potentially large numbers of partial activations. The growth of partial activations as a result of pattern ordering affects memory requirements, and, to a lesser extent, reset time. As the number of partial activations increases, the memory required and the reset time also increase.

Placing patterns that match volatile facts at the bottom of a rule LHS improves run-time performance. Placing patterns that match the least common facts in a system at the top of

a rule LHS reduces the number of partial activations observed. It may be difficult to use these methods in practice, however, since both of them depend on knowing the frequency with which certain facts appear in the system. In some cases, this may be readily apparent, but in other cases, especially where the form of the data may vary widely, these may not be practical. Long term statistical analysis of the system performance may be required to make use of these optimizations.

REFERENCES

1. "CLIPS PROGRAMMER'S GUIDE, VERSION 6.0, JSC-25012, NASA Johnson Space Center, Houston, TX, June 1993.

2. "CLIPS USER'S GUIDE, VERSION 6.0", JSC-25013, NASA Johnson Space Center, Houston, TX, May 1993.

3. "ILOG Rules C++ User's Guide, Version 2.0", ILOG Corporation, 1993.

4. Forgy, Charles, "Rete: A Fast Algorithm for the Many Pattern / Many Object Pattern Match Problem", ARTIFICIAL INTELLIGENCE, vol 19, 1982, pg 17-37.

5. Giarratano, Joseph and Gary Riley, EXPERT SYSTEMS: PRINCIPLES AND PROGRAMMING, PWS-Kent Publishing Company, Boston, MA, 1989.

6. Lee, Ho Soo, and Schor, Marshall, "Match Algorithms for Generalized Rete Networks", ARTIFICIAL INTELLIGENCE, Vol 54, No 3, 1992, pg 249-274.

7. Miranker, Daniel, TREAT: A NEW AND EFFICIENT MATCH ALGORITHM FOR AI PRODUCTION SYSTEMS, Morgan Kaufmann Publishers, Inc, San Mateo, CA, 1990.

8. Schneier, Bruce, "The Rete Matching Algorithm," AI EXPERT, December 1992, pg 24-29.

SIMULATION IN A DYNAMIC PROTOTYPING ENVIRONMENT: PETRI NETS OR RULES?

Loretta A. Moore and Shannon W. Price
Computer Science and Engineering
Auburn University
Auburn, AL 36849
(205) 844 - 6330
moore@eng.auburn.edu

Joseph P. Hale
Mission Operations Laboratory
NASA Marshall Space Flight Center
MSFC, AL 35812
(205) 544-2193
joe.hale@msfc.nasa.gov

ABSTRACT

An evaluation of a prototyped user interface is best supported by a simulation of the system. A simulation allows for dynamic evaluation of the interface rather than just a static evaluation of the screen's appearance. This allows potential users to evaluate both the look (in terms of the screen layout, color, objects, etc.) and feel (in terms of operations and actions which need to be performed) of a system's interface. Because of the need to provide dynamic evaluation of an interface, there must be support for producing active simulations. The high-fidelity training simulators are normally delivered too late to be effectively used in prototyping the displays. Therefore, it is important to build a low fidelity simulator, so that the iterative cycle of refining the human computer interface based upon a user's interactions can proceed early in software development.

INTRODUCTION

The Crew Systems Engineering Branch of the Mission Operations Laboratory of NASA Marshall Space Flight Center was interested in a dynamic Human Computer Interface Prototyping Environment for the International Space Station Alpha's on-board payload displays. On the Space Station, new payloads will be added to the on-board complement of payloads in ninety day increments. Although a payload starts its development and integration processes from two to four years before launch, a set of new payloads' displays are due every ninety days. Thus, this drives the need for an efficient and effective prototyping process. The functional components of a dynamic prototyping environment in which the process of rapid prototyping can be carried out have been investigated.

Most Graphical User Interface toolkits allow designers to develop graphical displays with little or no programming, however in order to provide dynamic simulation of an interface more effort is required. Most tools provide an Application Programmer's Interface (API) which allows the designer to write callback routines to interface with databases, library calls, processes, and equipment. These callbacks can also be used to interface with a simulator for purposes of evaluation. However, utilizing these features assumes programming language knowledge and some knowledge of networking. Interface designers may not have this level of expertise and therefore need to be provided with a friendlier method of producing simulations to drive the interface.

This research is supported in part by the Mission Operations Laboratory, NASA, Marshall Space Flight Center, MSFC, AL 35812 under Contract NAS8-39131, Delivery Order No. 25. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressly or implied, of NASA.

A rapid prototyping environment has been developed which allows for rapid prototyping and evaluation of graphical displays [2]. The components of this environment include: a graphical user interface development toolkit, a simulator tool, a dynamic interface between the interface and the simulator, and an embedded evaluation tool. The purpose of this environment is to support the process of rapid prototyping, so it is important that the tools included within the environment provide the needed functionality, but also be easy to use.

This paper describes two options for simulation within the dynamic prototyping environment: petri nets and rule-based simulation. The petri net system, PERCNET [3], is designed to be used as a knowledge-based graphical simulation environment for modeling and analyzing human-machine tasks. With PERCNET, task models (i.e., simulations) are developed using modified petri nets. The rule based system is a CLIPS [1] based system with an X windows interface for running the simulations. CLIPS executes in a non-procedural fashion making it ideal for representing random and concurrent events required by the simulation. Its C language-based design allows external communication to be programmed directly into the model. In order to compare the two approaches for simulation, a prototype of a user interface has been developed within the dynamic prototyping environment with both simulation architectures. This paper compares the two systems based upon usability, functionality, and performance.

ARCHITECTURE OF THE DYNAMIC PROTOTYPING ENVIRONMENT

There are four components of the Human Computer Interface (HCI) Prototyping Environment: (1) a Graphical User Interface (GUI) development tool, (2) a simulator development tool, (3) a dynamic, interactive interface between the GUI and the simulator, (4) an embedded evaluation tool. The GUI tool allows the designer to dynamically develop graphical displays through direct manipulation. The simulator development tool allows the functionality of the system to be implemented and will act as the driver for the displays. The dynamic, interactive interface will handle communication between the GUI runtime environment and the simulation environment. The embedded evaluation tool will collect data while the user is interacting with the system and will evaluate the adequacy of an interface based on a user's performance. The architecture of the environment is shown in figure 1.

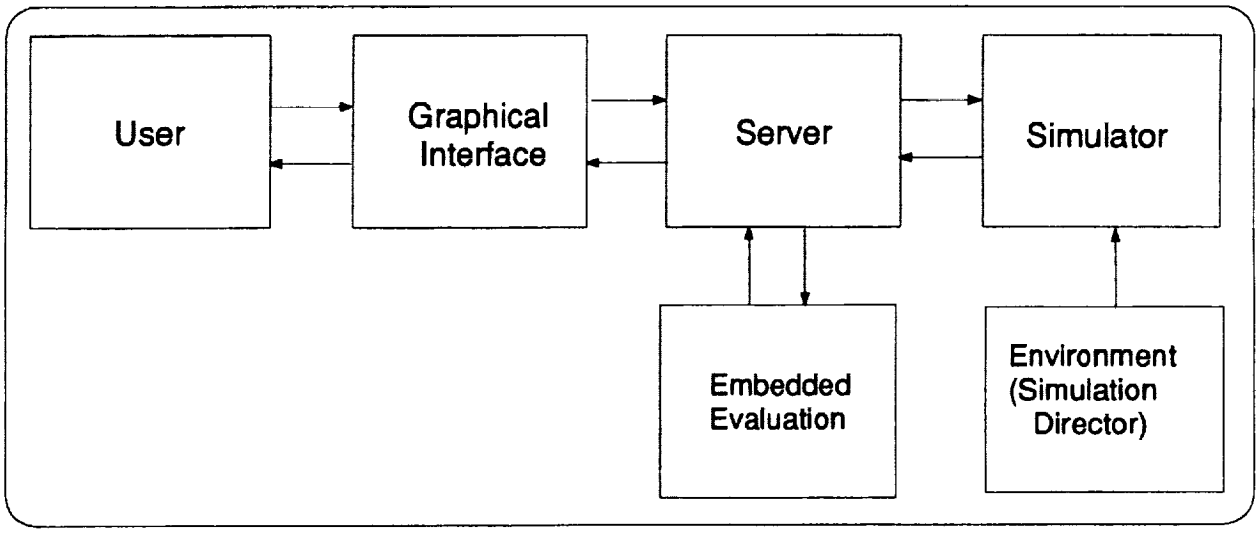


Figure 1 - HCI Prototyping Environment Architecture

Interface Development Tool

The Graphical User Interface (GUI) tool for the prototyping environment will allow the designer to create the display through direct manipulation. This includes the creation of static and dynamic objects, windows, menus, and boxes. The tool also allows objects created to be linked to a data source. During execution, the interface objects send and receive data and commands to the simulator by way of the data server. The user interface objects and their associated data access description are defined independent of the actual source of data. This first allows the development of the interface and the simulator to occur concurrently. Second, an interface developed with the GUI tool can later be connected to a high fidelity simulator and then to the actual flight software.

Simulator Development Tool

The simulator development tool provides the capability to develop a low fidelity simulation of a system or process. The development of a simulation has two important functions. First, the simulation helps the designer identify and define basic system requirements. Second, potential users can evaluate both the look (in terms of the screen layout, color, objects, etc.) and feel (in terms of operations and actions which need to be performed) of a system. The simulator provides realistic feedback to the interface based on user inputs.

Dynamic, Interactive Interface

This interface will handle communication between the GUI prototyping tool and the simulation tool during execution. The interface is a server which has been developed using the GUI's Application Programmer's Interface. Messages and commands can be sent and received both ways between the GUI and the simulator. The server also services requests from the embedded evaluation process, providing information as to which actions the user has taken and which events and activities have fired.

Embedded Evaluation Tool

An important aspect of the prototyping process is the ability to evaluate the adequacy of the developed graphical user interfaces. The embedded evaluation tool communicates with the server to receive information on the interaction between the user and the system. The types of data collected include user actions, simulator events and activities, and the times associated with these items. The collected data is analyzed to determine task correctness, task completion times, error counts, and user response times. The data is then analyzed to provide feedback as to which features of the interface the user had problems with and therefore need to be redesigned.

An Example: The Automobile Prototype

In order to assess the architecture described above a system was chosen to be prototyped in the environment. The system chosen for empirical evaluation of the HCI prototyping environment was an automobile. An automobile has sufficient complexity and subsystems' interdependencies to provide a moderate level of operational workload. Further, potential subjects in the empirical studies would have a working understanding of an automobile's functionality, thus minimizing pre-experiment training requirements.

An automobile can be considered a system with many interacting components that perform a task. The driver (or user) monitors and controls the automobile's performance using pedals, levers,

gauges, and a steering wheel. The dashboard and controls are the user interface and the engine is the main part of the system. Mapping the automobile system to the simulation architecture calls for a model of the dashboard and driver controls and a separate model of the engine. Figure 2 demonstrates how an automobile system could be mapped into the architecture described. The main component of the automobile is the engine which responds to inputs from the driver (e.g. the driver shifts gears or presses the accelerator pedal) and factors in the effects of the environment (e.g. climbing a hill causes a decrease in the speed of the car). The driver changes inputs to obtain desired performance results. If the car slows down climbing a hill, pressing the accelerator closer to the floorboard will counteract the effects of the hill.

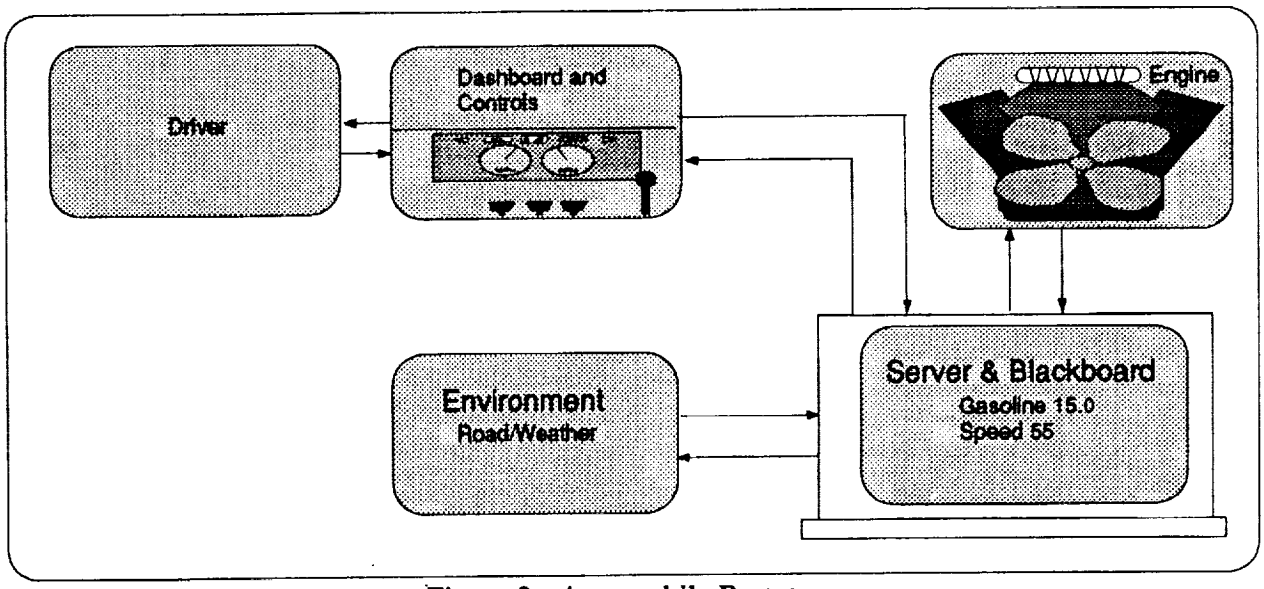


Figure 2 - Automobile Prototype

The dashboard and controls have been modeled using Sammi [5], a graphical user interface development tool developed by Kinesix. Two options have been investigated for simulation: petri nets and rules. Petri nets provide a graphical model of concurrent systems. The petri net system which has been used is PERCNET [3], developed by Perceptronics. PERCNET is designed to be used as a knowledge-based graphical simulation environment for modeling and analyzing human-machine tasks. With PERCNET, task models are developed using modified petri nets, a combination of petri nets, frames, and rules. The rule based system which has been used is CLIPS [1], a rule based language primarily used for the design of expert systems, developed by NASA. CLIPS executes in a non-procedural fashion making it ideal for representing random and concurrent events. The automobile system has been prototyped using both the petri net and rule-based systems as simulators and comparisons have been made based upon functionality, usability, and performance.

SIMULATION IN THE DYNAMIC PROTOTYPING ENVIRONMENT

Because of the need to provide dynamic evaluation of an interface rather than just static evaluation, there must be support provided for producing active simulation. Most GUIs, including Sammi, provide some sort of Application Programmer's Interface (API) which allow the developer to write call back routines which interface with databases, library calls, other processes and equipment. We would like to provide a means of building a low fidelity simulation of the system to drive the interface which requires little programming.

Basic simulation requirements include the ability to model events and activities, both sequentially and concurrently. The system should provide the ability to create submodels within the main model. The simulator clock must be linked to the system clock, and support should be provided for the creation of temporal events. The process must be able to communicate with UNIX processes using the TCP/IP protocol. Real-time communication must also be provided to allow the tool to communicate with the GUI tool on a separate platform via Ethernet. The ability for two-way asynchronous communication between the runtime versions of the interface and the simulator must be provided. The simulator must be capable of receiving data from the GUI tool to dynamically control temporal events, to modify the values of variables, and trigger events and activities. The ability to specify and send commands, data, and alarms to the GUI tool must also be provided. A simulator director should be able to send commands (e.g., start simulation, trigger scenario event, etc.) to the simulator from a monitoring station. An interface should be provided in order to bind interface objects to simulation objects in order to set the values of variables, trigger events or activities, and set temporal variables.

Simulation Using Petri Nets

PERCNET is a very powerful system analysis software package designed by Perceptronics, Inc. It provides an easy-to-use, graphical interface which allows users to quickly lay out a petri net model of the system. PERCNET uses "modified" petri nets, which allow each state to describe pre-conditions for state transitions, modify global variables, perform function calls and maintain a global simulation time.

Pictorially, Petri nets show systems of activities and events. Ovals represent activities which describe actions performed by the system. Activities are joined by events, represented by vertical bars, that occur during execution. Events are associated with user actions and environmental conditions. Execution is shown by tokens propagating through the system. Flow of control passes from activities to events. Before an event can fire all incoming arcs must have tokens. When this occurs, the event places tokens on all outgoing arcs passing control to activities. The behavior that an event exhibits during execution is dependant on the data contained in its frame. Frames record data related to each activity and event. Event frames may contain rules and functions. Activity frames allow the designer to specify a time to be associated with each activity. Figure 3 shows the top-level petri net of the automobile simulator.

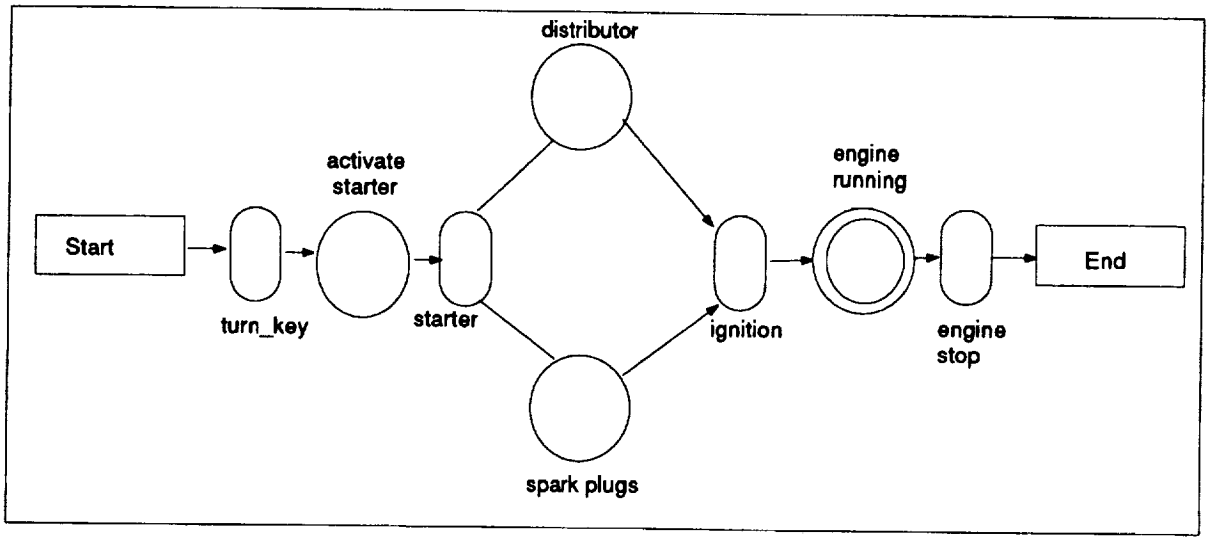


Figure 3 - Top-Level Petri Net of the Automobile Simulator

The starter is the component that is activated by the turning of the key. Before the starter can begin working, however, the key should be turned on, the driver must be wearing his/her seat belt, the car must be in neutral and the battery must have a sufficient charge to start the starter. When all three pre-conditions are true, the starter is activated and control advances to the right in the Petri net. Once the starter has been activated, it must do its part to start the automobile. The starter allows electricity to flow into the distributor where it is channeled into the spark plugs. As long as the starter is functioning, the distributor and spark plugs are activated. Finally, as long as the spark plugs and distributor are working properly and there is gasoline, the spark from the spark plugs ignites the gasoline mixture in the engine and ignition is achieved. Now that ignition has been accomplished, the engine is running. The concentric circles representing the engine_running activity in Figure 3 indicate that the state is shown in a sub-net.

The petri net representing the automobile passes from the ignition portion to the engine running state and remains in the running state until some condition causes the engine to stop running. The engine will stop running if the engine runs out of gas, runs out of oil, the temperature rises above a certain threshold, the key is turned off, the engine stalls (when the automobile is in some gear and the rpms fall below a threshold amount), the battery loses its charge or the fuel pump, oil pump, spark plugs or alternator fail.

The major components of the engine modeled are: fuel pump, oil pump, water pump, distributor, spark plugs, starter, battery, alternator, and fan. The condition of these components is modeled using a boolean variable indicating either that they are functioning or they are not. The boolean variables are then used as conditions within events occurring during the simulation. Details of the Petri Net implementation can be found in [2].

Simulation Using Rules

Since CLIPS is rule-based, it is completely non-procedural. Furthermore, it allows programmers to pick the strategy by which successive rule-firings are chosen. Certain rules may be designated fired by different priority levels (rules with the highest priority fire before rules with lower priority). Other rule-selection strategies govern how rules with equal priority are selected. Events and activities are represented by the pre- and post-conditions of rules. For example, the rule for activating the starter is:

```
(defrule TURN_KEY
                ?tick <- (clock_tick)
                (test (= 1 ?*key*))
                (test (= 1 ?*seatbelt*))
                (test (= ?*gear* 0))
                (test (> ?*battery* 10.0))
                (test (= ?*state* ?*READY*))
=>
                (bind ?*state* ?*STARTER*)
                (retract ?tick)
                (assert (clock_tick))
                (printout t "ACTIVATE STARTER (" ?*time* ")" crlf)
                (tick_tocks 2)
                (assert (updated TRUE))
)
```

In this project, CLIPS has been extended to include communication capabilities [4]. Two sockets have been provided for reading from and writing to the server. C functions have been developed to eliminate redundant information from the messages passed to the server. Another improvement compiled into the CLIPS executable has been a control process that allows a user to start, stop and quit CLIPS execution through a graphical interface.

The project also demonstrates some programming techniques used in CLIPS to support the simulation. A global simulation time should be maintained and a mechanism for keeping simulation execution time has been demonstrated. Another important feature that makes use of the timer is the periodic update feature. This ensures that CLIPS execution pauses (i.e., no rules may fire) every few seconds to send and receive information from the server. When this happens, control returns to the main routine which initializes communication with the server.

Writing CLIPS programs to take advantage of this strategy requires the incorporation of several techniques. These techniques include rules, variables, and functions which may be used in subsequent simulation designs. The first choice involves determining which values will be passed to or received from the server. All global variables (defined using the "defglobal" command) are passed to the server. No other values are passed. Facts and local variables may be used to store values which do not need to be passed to the server. It will be shown later how communication has been further streamlined for efficiency. The most important rule is the clock rule.

The clock rule stays ready at all times, but because the salience (i.e. priority) of the rule is kept low, it will not block the firing of other rules. When execution begins, the current system time is retrieved and stored. The current simulation time is always known by retrieving the system time and comparing it to the starting time. The new simulation time is temporarily stored in a variable called "new_time"and is compared to the last calculated time. If the two values are the same, then the clock rule has fired more than once within one second. In that case, the time is not printed and facts are reset to allow the clock rule to fire again.

A "clock_tick" fact is used in the preconditions of rules to allow them to become ready for firing. Without the clock_tick fact, a rule may never fire. Another time feature provided is the tick_tocks function. Often a programmer would like to force a rule to consume clock time. A call to the tick_tocks function forces execution to enter a side loop where the required time elapses before execution continues.

COMPARISON

Usability

Most features of PERCNET are easy-to-learn and use While some study of petri-net theory would benefit designers, much could be done with very minimal knowledge of petri-nets. One difficulty in working with PERCNET was the lack of available documentation on the Tool Command Language (TCL). All function calls, calculations, communication and ad-hoc programming are done using this language. Perceptronics provides only minimal documentation on the use of the language within PERCNET making it very difficult to perform anything more than the most basic operations. However, PERCNET's graphical interface is very appealing to users.

CLIPS is a rule-based language, which means that there may be a larger learning curve than there is with PERCNET's point-and-click interface. After the initial learning stages, however, CLIPS leaves a developer with an immensely powerful simulation tool. The main advantage is flexibility.

CLIPS was written in the C programming language and is completely compatible and extendible with C functions. Knowing C in advance can significantly lessen the learning curve. Many of the "non-C" features of CLIPS resemble LISP. CLIPS has been a tremendous surprise to work with. A basic proficiency with CLIPS may be gained quickly and one can learn to do very useful things with the language. Writing the rules for the simulation was actually the easiest part of the project. As proficiency with the language developed, more advanced features provided tremendous possibilities. The manuals present the language in a very easy to read format, contained extensive reference sections and sample code. Furthermore, the manuals outline how CLIPS may be easily extended to include C (and other) functions written by programmers.

Functionality

As this project began, PERCNET was a closed package, that is, there was no provision for communicating with other applications. NASA contracted Perceptronics to modify PERCNET to allow for such a feature. The final result was a revision of PERCNET which would allow communication with other applications through the use of sockets. Applications are allowed to request that global variables be retrieved and/or modified. PERCNET essentially opened it's blackboard (i.e., global data store) to other applications. The other application in this case being the server.

After several functions were added to CLIPS (see descriptions in previous sections), the CLIPS system performed the same functions as the Petri Net simulator. If a new system is prototyped, the only changes which would be needed are to the knowledge base. The communication link developed for the Sammi-CLIPS architecture uses the blackboard paradigm to improve modularity, flexibility, and efficiency. This form of data management stores all information in a central location (the blackboard), and processes communicate by posting and retrieving information from the blackboard. The server manages the blackboard, allowing applications to retrieve current values from the board and to request that a value be changed. The server accepts write requests from valid sources and changes values. The comparison of the two architectures goes much further than comparing the two simulation designs. The design of the communication link significantly affects the flexibility and performance of the architecture.

Performance

The performance within the Petri Net architecture was not acceptable for real-time interface simulation. Interfaces running within this architecture exhibit a very slow response rate to user actions when PERCNET is executing within its subnets. The PERCNET execution is also using excessive amounts of swap space and memory which also affect the refreshing of displays.

Early analysis attempted to find the exact cause of the poor performance; however, only limited work could be done without access to PERCNET's source code. Since PERCNET's code was unavailable, we could only speculate about what was actually happening to cause the slow responses. It was determined that the cause of much of the problem was that PERCNET was trying to do too much. In the PERCNET simulation architecture, PERCNET is actually the data server for the environment. The global blackboard is maintained within PERCNET. The server only provides a mechanism for passing information between PERCNET and other applications. The server is connected to PERCNET by a socket and the server is actually on the "client" end of the connection-oriented socket. The server establishes connections with PERCNET and Sammi and then alternately receives information from each. Any data or commands received from Sammi are passed immediately to PERCNET. Commands from PERCNET for Sammi are passed immediately through, as well. Finally, the server sends Sammi copies of all variables. Since PERCNET is the blackboard server, as well as the simulator, PERCNET's performance would naturally be affected by the added burden.

Lastly, the method provided for sending variables to the server was terribly inefficient. When a calculation was performed in the simulation model for a variable that was needed by the interface, that variable was passed to the server whether or not it's value had changed from the previous iteration. No mechanism was provided for restricting the number of redundant values passed across the communication link. As a result, PERCNET passed every value back to the server when only a few had actually changed.

Each of these limitations was addressed in the design of the server and blackboard in the rule-based architecture. The server program may be divided into three portions: blackboard management, Sammi routines, CLIPS routines. The Sammi and CLIPS routines are provided to communicate with the respective applications. These routines map data into a special "blackboard entry" form and pass the data to the blackboard management routines. The blackboard routines also return information to the Sammi and CLIPS routines for routing back to the applications. The blackboard management routines require that each application (many more applications may be supported) register itself initially. Applications are assigned application identification numbers which are used for all subsequent transactions. This application number allows the blackboard to closely monitor which variable values each application needs to see. It also provides a mechanism for installing a priority scheme for updates.

The overwhelming advantage of the CLIPS and blackboard combination is the flexibility and potential they provide. Features are provided that allow modifications which can affect performance. The ability to tune the performance has allowed the simulation architecture to be tailored to specific running conditions (e.g., machine limitations, network traffic and complexity of the interface being simulated). Several parameters may be modified to alter performance. Tuning tests have improved performance. More detailed performance testing is planned to verify the results.

CONCLUSION

The goal of the architecture has been to provide simulation of user interfaces so that they may be designed and evaluated quickly. An important portion of the dynamic prototyping architecture is therefore the simulator. Ease-of-use is very important, but performance is critical. The Petri Net architecture's ease-of-use is currently its only advantage over the Rule-Based architecture. The Rule-Based design overcomes this with power and flexibility. Work currently in progress include a detailed analysis of the performance of the communication link and a design of a graphical interface to CLIPS.

REFERENCES

1. CLIPS Reference Manual, NASA Johnson Space Flight Center, Houston, Texas, 1993.

2. Moore, Loretta, "Assessment of a Human Computer Interface Prototyping Environment," Final Report, Delivery Order No. 16, Basic NASA Contract No. NAS8-39131, NASA Marshall Space Flight Center, Huntsville, Alabama, 1993.

3. PERCNET User's Manual, Perceptronics Inc., Woodland Hills, California, 1992.

4. Price, Shannon, "An Improved Interface Simulation Architecture", Final Report for Master of Computer Science and Engineering Degree, Auburn University, Auburn, Alabama, 1994.

5. Sammi API Manual, Kinesix Corporation, Houston Texas, 1992.

OMIT

Session 6A: Design Applications

Session Chair: Carol Redfield

COLLABORATIVE ENGINEERING-DESIGN SUPPORT SYSTEM

Dong Ho Lee and D. Richard Decker

Electrical Engineering and Computer Science Department
19 Memorial Drive West
Lehigh University
Bethlehem, PA 18015

ABSTRACT

Designing engineering objects requires many engineers' knowledge from different domains. There needs to be cooperative work among engineering designers to complete a design. Revisions of a design are time consuming, especially if designers work at a distance and with different design description formats. In order to reduce the design cycle, there needs to be a sharable design description the engineering community, which can be electronically transportable. Design is a process of integrating that is not easy to define definitively. This paper presents Design Script which is a generic engineering design knowledge representation scheme that can be applied in any engineering domain. The Design Script is developed through encapsulation of common design activities and basic design components based on problem decomposition. It is implemented using CLIPS with a Windows NT graphical user interface. The physical relationships between engineering objects and their subparts can be constructed in a hierarchical manner. The same design process is repeatedly applied at each given level of hierarchy and recursively into lower levels of the hierarchy. Each class of the structure can be represented using the Design Script.

INTRODUCTION

Design is a fundamental purposeful human activity with a long history. Design can include creative artistic and engineering components. Knowledge-based design systems deal with factors that tend to be limited to the engineering aspects of design. Many researchers have developed knowledge-based engineering design systems. Many of these systems were developed for the specified design domain using their own unique problem solving method [2, 5, 9, 12, 21]. Some have tried to develop domain independent knowledge-based design systems. The DIDS (Domain Independent Design System) by Runkel [18] was developed as set of tools that can provide a configuration-design system from a library. DOMINIC [10] treats design as best-first search by focusing on the problem of iterative redesigning of a single object. GPRS (Generative Prototype Refinement Shell) was developed by Oxman [15], which used Design Prototype [8, 22] as a knowledge representation. In the real world, most engineering designs are so complex that a single individual cannot complete them without many other engineers' help. Cooperation between different engineering designers is not a simple process because each designer may have a different perspective for the same problem, and multiple revisions of a design are needed in order to finish a project. Designers may have different interpretations of the same design value or may want to access special programs to determine values for the design variables in which they are interested. In order to achieve the above goals, there needs to be a design knowledge

representation that can be shared between designers and that can be modified to the designers' needs. In addition, if a designer needs to execute a special program, the design system should provide a scheme to do so. This paper describes a Design Script as an abstract model of the design process that is based on hierarchical design structure and shows how to capture design knowledge and integrate data and tools into a knowledge based design system.

MODELS OF DESIGN PROCESSES

Dieter [7] describes the design process in his book as follow: "There is no single universally acclaimed sequence of steps that leads to a workable design." But it is possible to make the fundamental design process as simple as an iterative process of analysis, synthesis, and evaluation (Fig. 1). Analysis is required to understand the goals of the problem and to produce explicit statements of functions. The synthesis phase involves finding plausible solutions through the guidance of functions that are produced from the analysis phase. The evaluation process checks the validity of solutions relative to the goals. The evaluation phase can be divided into two different types of jobs. One is to compare the solution with existing data if the solution is composed of comparable data; and the other is to compare the solution values derived from the current design solution through simulation process executions with the given goals.

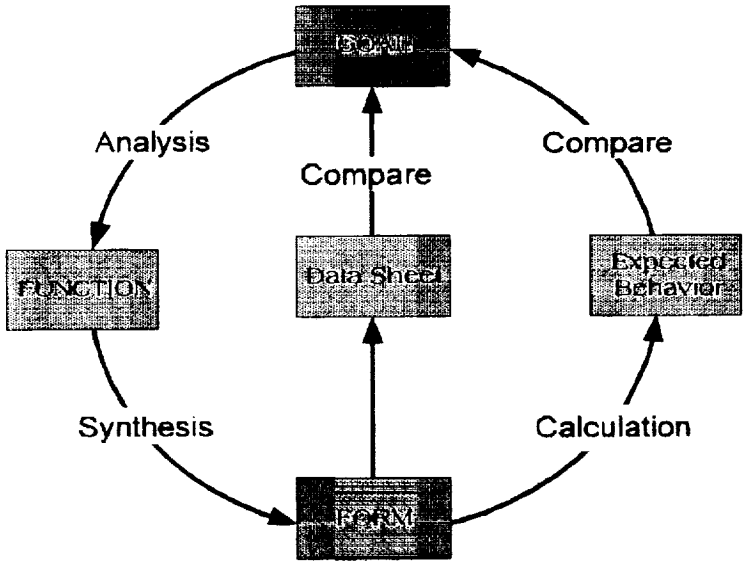


Fig. 1. Goal, Function, Form and Behavior Relationship

Problem decomposition is a well-known problem solving strategy in knowledge-based design systems. Once a complex design problem (a complete object) is decomposed into simple design problems (subparts), it is much easier to handle. The engineering design object is considered as being composed of sub-parts in a hierarchical structure. In other words, the problem of designing a complete object is comprised of designing number of subparts recursively until the designing process reaches the elemental subparts. The extent and complexity of a design problem can be different at different levels of hierarchy, but the identical design process can be applied at all hierarchical levels.

DESIGN SCRIPT

The DS (Design Script) is presented in this paper as an engineering design knowledge representation scheme. The DS is implemented in COOL (CLIPS Object Oriented Language) of NTCLIPS which is a Windows-NT version of CLIPS developed by the authors [20]. The common design process which is composed of analysis, synthesis, and evaluation phases can be abstracted by encapsulating design activities, such as goal decomposition and invoking methods. The abstracted model Design Script is placed at the top of the design knowledge for a domain. The major components of the DS are the goals, functions, and form of the part which is designed (Fig. 2). These abstractions include how the goals are decomposed and passed to lower level objects, how the functions of subparts are represented and used, and how explicit knowledge about reasoning and transforming process are formed and used. The instances of design sub-part can inherit these abstractions.

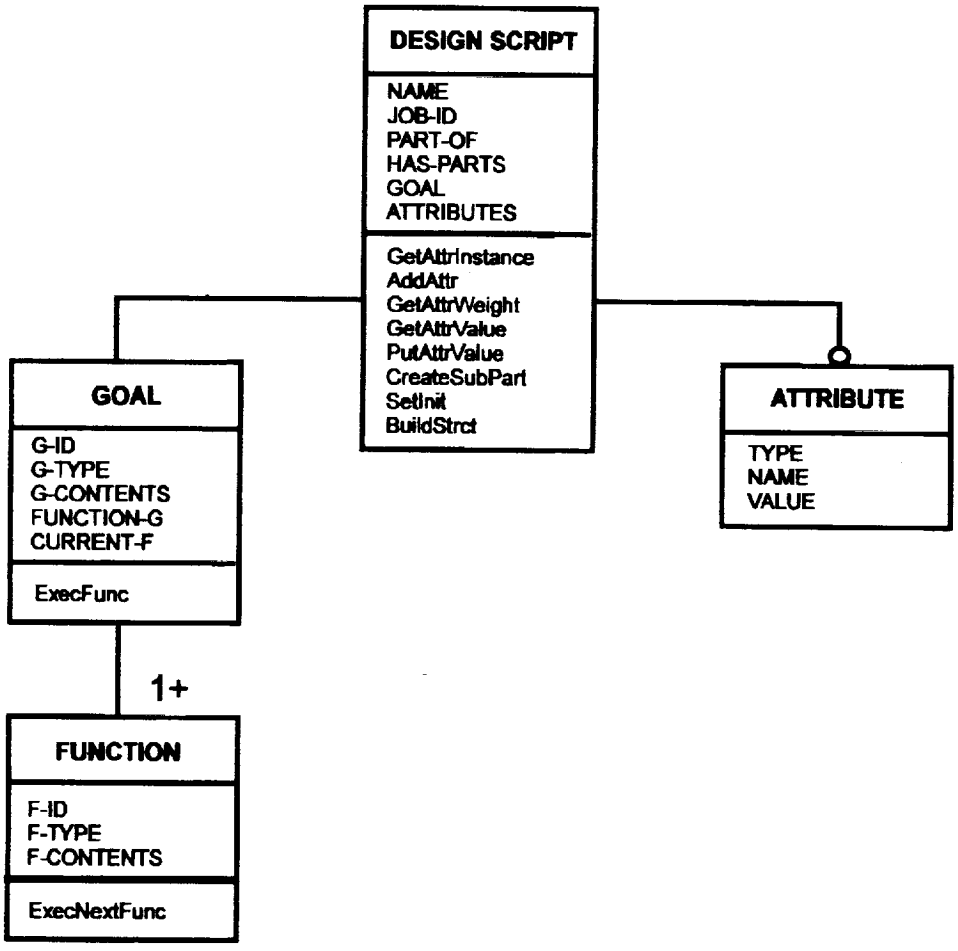


Fig. 2. Main Structures of Design Script

Usually, the initial goal of the design object is represented using natural language rather than by using a technical representation such as "power amp can drive as low as 2-ohm impedance loud speaker with 80 dB SPL." Then the goal is refined to the functions such as "require an amplifier

which can provide 20A current at peak with at least 200 W/ch power." The functions of the initial stage of the design process are regarded as goal of the next stage of design. The relations between goal and its functions are acquired knowledge from the designer's point of view. When a designer sets goals (or requirements) for the design object, the designer has what should be required to meet the goals. Although, the relations between goals and functions are intrinsic knowledge, they should be represented explicitly in the knowledge-based design process. The ***Goal*** slot of DS will hold a bound instance of the ***Goal-Design*** class as a slot value. Each instance of the ***Goal-Design*** class has its intrinsic functions kept as a multi-value slot to meet current goals. The ***Attributes*** slot of DS is a multi-value slot which keeps all the design variables and values for the design problem. DS provides various manipulation methods such as read-out and write-in the value of the attribute, decomposing the design into sub-parts, and creating instances of attributes and sub-parts. There always exist tradeoffs between generality and efficiency. To improve the efficiency of the design system, it could be built as a very domain specific. On the other hand, to make it general may require a sacrifice of efficiency. The DS has been developed to be very general, in that it can be applicable to different design domains.

GENERALIZATION AND AGGREGATION

Generalization is a powerful abstraction that can share commonalties among classes. Usually the generalization is represented as a superclass. The Design Scrip is a generalization of the design process that is based on the hierarchical structure of engineering objects. Each subclass inherits the features of its superclass. The inheritance mechanism of most object-oriented programming languages is based on a ***sub-type*** or ***is-a*** relation between classes. The COOL of CLIPS is not an exception. All the parts of a design object are represented as a subclass of DS so that they can inherit the properties of DS by using the built-in ***is-a*** hierarchy relation. But, the hierarchy relations among the parts themselves can not be represented by using ***is-a*** or ***kind-of*** relations, because they are actually ***part-of*** hierarchies (Fig. 3). Aggregation is the ***has-part*** or ***part-of*** relationship in which objects are part of the whole assembly. For example, let's take an example of a SO-8 package. We can say "*An SO-8 package is a kind of IC package.*" and "*IC is a kind of electronic component.*" ("SO-8" stands for "Small Outline 8 pin" and "IC" stands for "Integrated Circuit") We can see the ***is-a*** hierarchy relations between *an SO-8 package*, *IC package*, and *electronic component.* If we build the hierarchical structure for *SO-8*, we can see that it is composed of a mold material, a lead frame, wire bonds, etc. The lead frame is composed of lead1, lead2, die pad, etc. The relationship between these parts can not be represented with ***is-a*** relations. We can say the lead frame is not a type of SO-8, but a sub-part of the SO-8. So, in order to express the design knowledge, a mechanism must be available to handle the ***part-of*** relationship hierarchy. The slots ***PART-OF*** and ***HAS-PART*** of DS are used to define the ***part-of*** relations relative to super and subparts of the part under consideration.

DESIGN KNOWLEDGE MANAGEMENT

Many design engineers have stressed the uselessness of unmodifiable design knowledge. If fixed engineering design knowledge doesn't have a self-adaptive function for new situations, it may not work properly in these cases. For our system, the DS needs to follow the syntax of CLIPS when building a design knowledge base for a specific application design domain, because DS is

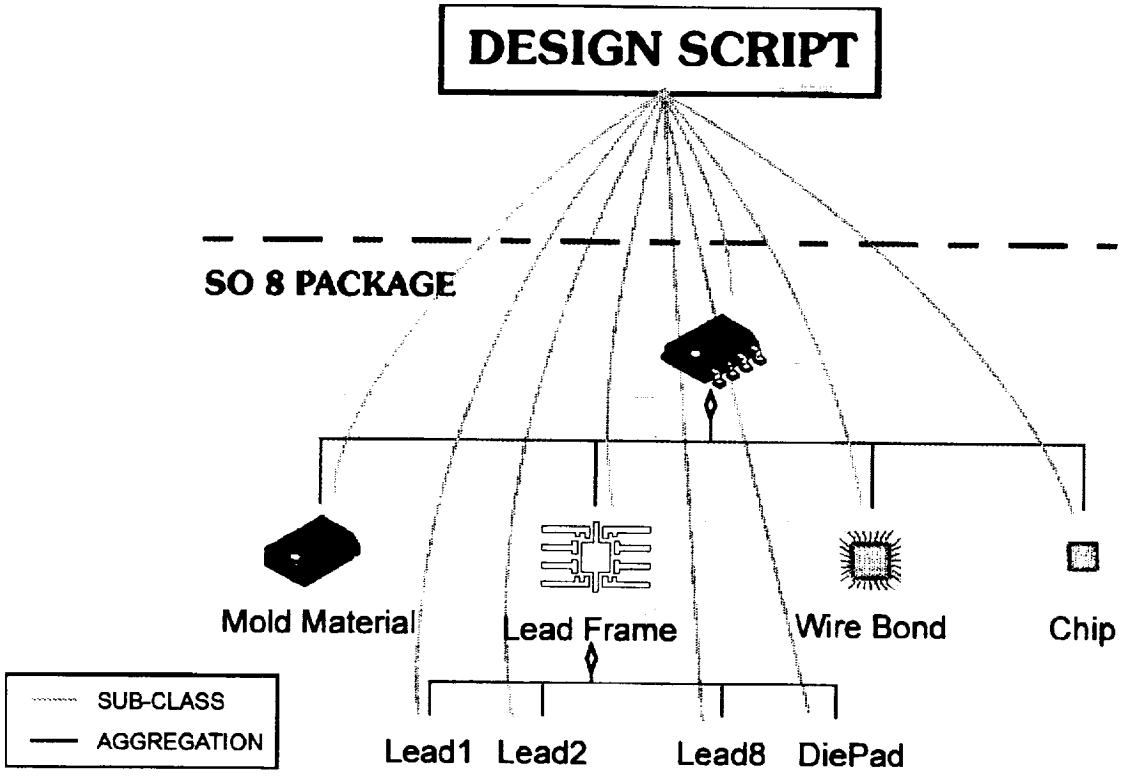


Fig. 3. Hierarchical Structure of SO8 IC Package and Design Script

implemented in CLIPS. The syntax of CLIPS is similar to that of the LISP language which uses lots of parentheses. CLIPS uses prefix notation in numerical computations rather than infix notation, which is generally used to represent algebraic expressions. If the developer is not familiar with this type of environment, it could be difficult to build the knowledge base. Considering that most design engineers are not computer scientists who can easily adapt to new programming environments, a user interface for knowledge entry and building is provided as a most essential part of the design system. The DS provides a GUI (Graphical User Interface) empty form as part of its functionality (Fig. 4). The design knowledge can be built easily by using the knowledge entry dialog-box on a part by part basis. What the designer has to do is fill in each field of the dialog box with the necessary information for designing or analyzing the object or subpart. It is an essential process to specify the design object in the hierarchical structure before entering design knowledge. Each dialog-box can edit design knowledge for a single (composite or elemental) part, such as its part-of and has-part relations, its goal, functions of the goal, design attributes for the part, and its numerical methods to get the implicit values of its design variables. When the user edits the dialog-box, the contents of the fields are stored temporarily in instances of the ***KENTRY*** class. Because the purpose of the ***KENTRY*** class is to provide an editable form of design knowledge that is easy to use, the contents of the design knowledge are kept as text in the corresponding slot. Once this information is entered by the user, it can be saved into a file in the form of instances of the ***KENTRY*** class from the main menu. Later, the user can modify the design knowledge by loading a previously created version

Knowledge Entry DialogBox

Object-Oriented Data in File

Fig. 4 Design System Structure with Knowledge Entry Dialog Box

from the file. Whenever the user wants to run the design system with the current design knowledge, he first compiles the knowledge from the instance form to the form of classes and their message-handlers which are in CLIPS syntax.

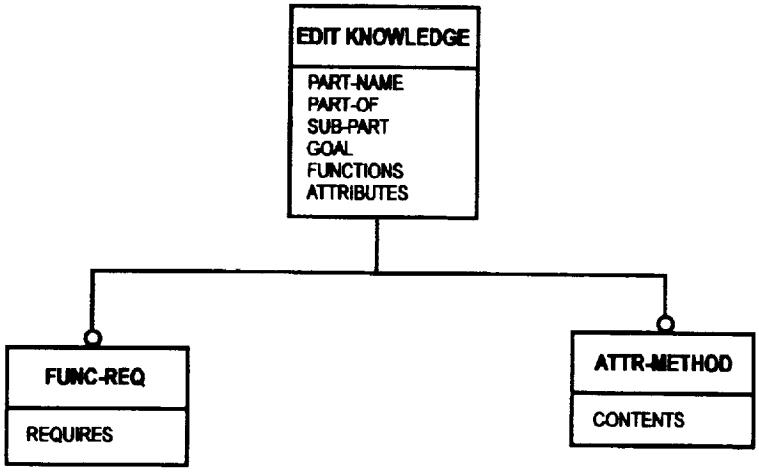


Fig. 5. Knowledge Entry Class

DESIGN APPLICATION: LEAD-FRAME DESIGN FOR PQFP

The DS has been applied in the domain of Lead-Frame design for a plastic quad flat pack IC package (PQFP). Acquiring design knowledge for the lead-frame of a PQFP is not a simple process. First of all, manufacturers tend to use their own knowledge to produce packages for their chips. Moreover, each company wants to keep this knowledge as proprietary information. There is available a limited amount of public design knowledge such as the standard package outlines contained in the *JEDEC Package Standard Outline*, research paper [11], or handbooks [19, 23]. This standard is widely used throughout the community of semiconductor manufacturers. The JEDEC standard is just for the outside dimensions, such as body size, length of outside lead, lead pitch, etc. The JEDEC manual doesn't refer to the inside dimensions of the package. This limitation reveals one difficulty of standardization in the packaging world. The major goal of the lead-frame design system for PQFP is to define the geometry of a lead-frame which can satisfy the JEDEC standard and the electrical and mechanical design constraints.

Fig. 6 describes the design knowledge for a lead-frame in graphical format. The values of lead-pitch, lead-width, and outside lead length are decided from JEDEC standard dimensions which are stored in a JEDEC database. The size of the chip bonding pad is decided by reference to the chip design technology base in use. The maximum wire span of the bonding wire from the bonding pad on the chip to the lead tip of the lead-frame is 100 mil. The lead tip pitch is around 10 mil. The width of the lead tip is around 6 mil. The length of the lead tip is dependent on its position. Each lead has an anchor hole. The fineness of the lead tip is also limited by the sheet metal blanking technology. The whole area of the metal part inside the plastic package is equal to or a slightly less than half of the entire plastic area.

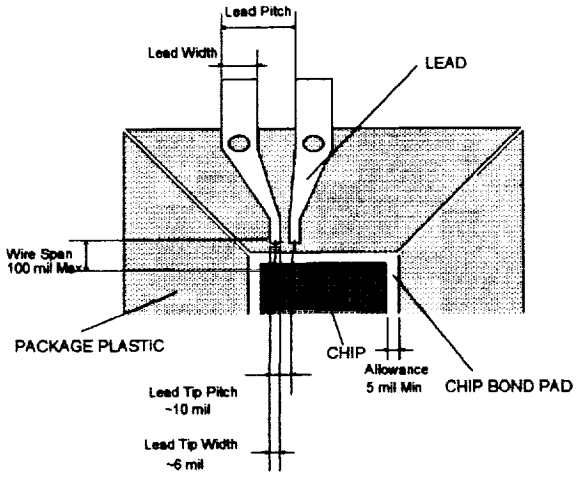


Fig. 6. Design Knowledge for the Lead-Frame of the PQFP.

When a user loads or edits the design knowledge using the *Knowledge Entry* dialog box and compiles it, the design knowledge is translated into CLIPS syntax (Fig. 7.). The first part of the knowledge is for the definition of LF-PQFP (Lead Frame of Plastic Quad Flat Pack) class which has *GOAL* and *HAS-PART* slots. The default value of the *GOAL* slot is the name of an instance of the GOAL-DESIGN class which has information about how to achieve the goal. Another multi-value slot contains the name and number of the subpart class. The sub-part of the lead-frame of the PQFP is composed of one die-bond-pad and the number of I/O leads in one octant of the package.

```
(defclass LF-PQFP (is-a DESIGN-SCRIPT)
 (role concrete)
 (slot GOAL (create-accessor read-write)(default "LF-1stGoal"))
 (slot WIRE-SPAN (create-accessor read-write)(default 100))
 (multislot HAS-PART (create-accessor read-write)(default "DieBondPad" "Lead1" "Lead2"
    "Lead3" . . . "Lead11"))
)

(message-handler LF-PQFP set-attributes()
 (bind ?self-i (instance-name ?self))
 (bind ?ins-name (symbol-to-instance-name (sym-cat ?self-i "LEADTIPPITCH")))
 (make-instance ?ins-name of AN-ATTRIBUTE
  (ATTR-NAME "LEAD-TIP-PITCH")
  (value 0.254))
 (send ?self add-attr ?ins-name)

(message-handler LF-PQFP decide-Die-Size ()
 (bind ?SB (send ?self get-attr-value (send ?self get-attr-instance "Die-SB"))
 (bind ?Nio (send ?self get-attr-value (send ?self get-attr-instance "N-IO"))
 (bind ?Pd (send ?self get-attr-value (send ?self get-attr-instance "PAD-PITCH"))
 (bind ?DieSize (+ (* 2 ?SB) (*?Pd (+ 1 (/ ?Nio 4)))))
 (send ?self put-attr-value (send ?self get-attr-instance "PAD-PITCH") ?DieSize)
)
```

Fig. 7. CLIPS syntax for the Class and Message-Handler.

The reason for designing only an octant of I/O leads is that the geometry of the remaining leads can be derived from this portion of the geometry by considering the symmetry of the lead-frame. The next message-handler takes care of creating and storing the instances of the attribute. The last message-handler is an example of a method used to decide the design value of the attribute.

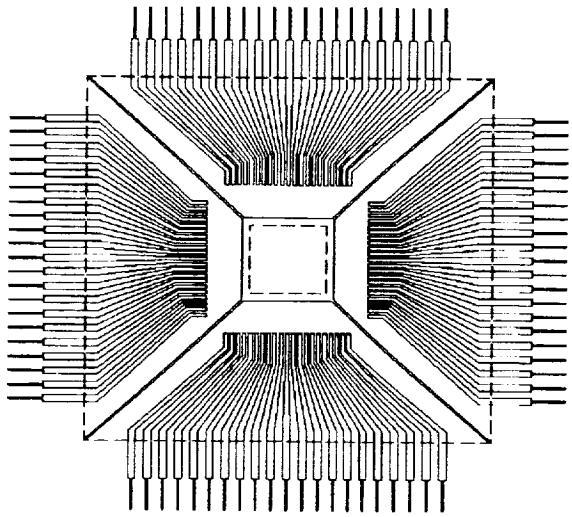


Fig. 8. Design Result: Lead-Frame of an 84 pin PQFP

Figure 8 shows the lead-frame of an 84 pin PQFP that was created by running the above lead-frame design system. The outer dotted square represents the plastic body of the PQFP and the inner dotted square shows the minimum size of the chip. The figure is drawn using the GNU plot program by providing the geometry that is produced by the lead-frame design system.

CONCLUSIONS AND FUTURE WORK

The main contribution of this research is in providing a general design-process control framework and a general design-knowledge representation scheme for physical objects to be designed using knowledge-based design systems. The Design Script developed here can be applied in any design domain because it contains domain knowledge independent about the design process. Usually, design is not only an individual activity but also requires cooperative team work that involves a number of designers from different fields. To support cooperative design work, DS provides an excellent framework that can be used in different domains. The capability of DS has been demonstrated in the domain of leadframe design for PQFPs.

This work is a part of the ongoing project. Much remains to be done to enhance the functionality of DS. Especially, management of design knowledge is a good candidate for future work. The current knowledge entry process doesn't provide any debugging functions. The design knowledge may be easily broken without careful knowledge entry. But there is also a need for a version management method like RCS (Revision Control System) or Aegis to enhance the productivity of cooperative design work. This enhancement is being considered to provide such functionality to the DS.

ACKNOWLEDGMENTS

The financial support from Semiconductor Research Corporation under the contract number MP-071 is gratefully appreciated.

REFERENCES

1. Akagi, S., "Expert System for Engineering Design Based on Object-Oriented Knowledge Representation Concept," *Artificial Intelligence in Design (ED. PHAM D. T.)*, 1991, 61-95.
2. Brown, D. C., and Chadrasekaran, B., "Design Problem Solving: Knowledge Structures and Control Strategies," *Pitman/Morgan Kaufmann*, 1989.
3. Chandrasekaran, B., "Generic Tasks in Knowledge-Based Reasoning: High-Level Building Blocks for Expert System Design," *IEEE Expert*, 1986, 1(3), 23-30.
4. Chandrasekaran, B., "Design Problem Solving: A Task analysis," *AI Magazine*, 1990, 11(4), 59 - 71.
5. Choi, C-K., and Kim, E-D., "A Preliminary Model of I-BUILDS: An Intelligent Building Design System," *Knowledge Based Expert Systems in Engineering: Planning and Design (Ed. Sriram D. and Adey R.A.)*, 1987, 331-343.
6. Coyne, R. D. et al., "Knowledge-Based Design Systems," *Addison-Wesley Publishing Co.*, 1990.
7. Dieter, G. E., "Engineering Design A Materials and Processing Approach," McGraw-Hill Book Co., 1983.
8. Gero, J. S., "Design Prototypes: A Knowledge Representation Schema for Design," *AI Magazine*, 1990, 11(4), 26-36.
9. Harty, N., "An Aid to Preliminary Design," *Knowledge Based Expert Systems in Engineering: Planning and Design (Ed. Sriram D. and Adey R. A.)*, 1987, 377-392.
10. Howe, A. E. et al., "Dominic: A Domain-Independent Program for Mechanical Engineering Design, *Artificial Intelligence in Engineering Design*, July 1986, 1(1), 23-28.
11. Jahsman, W. E., "Lead Frame and Wire Length Limitations to Bond Densification," *Journal of Electronic Packaging*, December 1989, Vol. 111, 289-293.
12. Maher, M. L., "HI-RISE and Beyond: Directions for Expert Systems in Design," *Computer Aided Design*, 1985, 17(9), 420-427.
13. Maher, M. L., "Process Models for Design Synthesis," *AI Magazine*, 1990, 11(4), 49-58.
14. Nguyen, G. T. and Rieu D., "SHOOD: A Design Object Model," *Artificial Intelligence in Design '92 (Ed. Gero J. S.)*, 1992, 221-240.
15. Oxman, R., "Design shells: a formalism for prototype refinement in knowledge-based design systems," *Artificial Intelligence in Engineering Design Progress in Engineering Vol.12*, 1993, 92-98.

16. Oxman, R., and Gero J. S., "Using an Expert System for Design Diagnosis and Design Synthesis," *Expert Systems*, 1987, 4(1), 4-15.

17. Rumbaugh, James et al., Object-Oriented Modeling and Design, Prentice Hall, 1991.

18. Runkel, J. T., et al., "Domain-Independent Design System, Environment for rapid development of configuration-design systems," *Artificial Intelligence in Design '92 (Ed. Gero J. S.)*, 1992, 21-40.

19. Seraphim, D. P.(Ed.), Principles of Electronic Packaging, *McGraw-Hill*, 1989.

20. STB, CLIPS Reference Manual Ver. 6.0, *Lyndon B. Johnson Space Center*, 1993.

21. Sriram, D. et al., "Knowledge-Based Expert Systems in Structural Design," *Computers and Structures*, 1985, 20(13) : 1-9.

22. Tham, K. W. and Gero, J. S., "PROBER -A Design System Based on Design Prototypes," *Artificial Intelligence in Design '92 (Ed. Gero J. S.)*, 1992, 657-675.

23. Tummala, R. R.(Ed.), Microelectronics Packaging Handbook, *Van Nostrand Reinhold*, 1989.

5/1-61
34094
P-7

N95-19758

CHARACTER SELECTING ADVISOR FOR A ROLE-PLAYING GAME

Carol L. Redfield, Ph.D.
University of Texas at San Antonio
6900 North West Loop 1604 West
San Antonio, TX 78249
CRedfield@SwRI.edu
(210) 679-7625

Felicia Berlanga
5510 Chancellor
San Antonio, TX 78229
(210) 684-8168

ABSTRACT

Role-playing games have been a source of much pleasure and merriment for people of all ages. The process of developing a character for a role-playing game is usually very, very time consuming, delaying what many players consider the most entertaining part of the game. An expert system has been written to assist a player in creating a character by guiding the player through a series of questions. This paper discusses the selection of this topic, the knowledge engineering, the software development, and the resulting program that cuts the time of character development from about 4 hours to 30 minutes. The program was written on a PC and an Apollo in CLIPS 4.3 and currently runs on the Apollo.

INTRODUCTION

The 1993-1994 Independent Study Mentorship (ISM) program through the Northside Independent School District in San Antonio started in the Summer of 1993. The ISM program allows students to study an area of their own choosing for a whole school year. During the ISM year, the student is paired with a mentor from the community who will guide the student in his or her studies. At the end of the ISM year, the student must have developed a tangible product utilizing the knowledge he or she has acquired throughout the year. The Young Engineers and Scientists (YES) program is for ISM students interested in the areas of engineering and applied sciences. YES gives students a chance to see what actually is done in careers within the engineering and sciences. The YES program consists of two components: a three week intensive study at Southwest Research Institute (SwRI) and a one-on-one mentorship through the school year. In YES, the mentors are selected from the staff at SwRI, and they mentors are supported financially by the NSF grant that funds the YES program. In most other mentorship programs, the mentor's time is provided on a volunteer basis.

Felicia was a YES program student who was interested in artificial intelligence, specifically expert systems, which was Carol's expertise. Together we designed an expert system to help role-players build their characters more efficiently and with much less time required. In this paper, the steps taken in developing the system are outlined including product development, research, programming, and conclusions.

PRODUCT DEVELOPMENT

After deciding upon developing an expert system, an appropriate topic had to be selected. Since Carol's Ph.D. work was in game playing and Felicia had an interest in role-playing games, we decided to work in the domain of role-playing games. When we looked into the current state of the gaming world, we saw that a tremendous amount of time is required in making a character for many role-playing games. Role-playing consists of assigning traits according to a set of rules to a character which the role-player will use to act out all actions as if the player were that character within a story. To make a specific desired character, a player typically takes time to make sure the character will be described well on paper and play well during the game. We decided to develop an expert system to assist in character development so that the process of developing a character would take much less time and would hopefully develop a better character with which to play the game. One way to develop a character is by asking a series of multiple choice questions of the player. Each response by the player would further develop and define the character and would lead to the next set of questions based on the previous response. The questions were developed in such a way so as to have the character be one that closely matches the qualities and characteristics that the player is picturing about their character. This question development is where much of the research was done in addition to determining a good way to implement the design of the system.

RESEARCH

The White Wolf Storyteller game called Mage was selected because of its popularity and availability of experts in playing the game and especially in developing characters. We designed our system to be a series of questions of the user to determine the characteristics preferred by the player. We first gathered information and performed our knowledge engineering by observing games being played and interviewing the players about character generation. A few local role-players who knew the game and played it were willing to be our experts. A number of multiple choice questions were developed in interviewing the expert game players.

The Mage character sheet shown on the next page is used in character development and play of the game. It is broken up into many sections. The first section consists of the character's name and other components that make up the character's basic identity. The second section, *attributes*, pertains to the character's mental, social, and physical characteristics. The third section, *abilities*, defines a character's talents, skills, and knowledge or the things they can do. The fourth section, *advantages*, deals with the spheres of magic the character can obtain or use. The last section contains the character's health levels, combat weapons, and other characteristics that have to do with the character's longevity. This last section and the advantages section are not developed in the expert system. The Mage Character Advisor walks a player through selecting the attributes and abilities for the character being developed.

On the Mage sheet, each circle by a characteristic represents a point for that area when it is filled. The point levels indicate how much of that characteristic or ability the character possesses, and in turn how many dice can be used in playing the game. Players can lose or gain points during the course of a game, and during games the Gamemaster (GM), who is the person who runs the game, gives out experience points which can be used to buy more points.

MAGE

The Ascension

Name:	Essence:	Tradition:
Player:	Nature:	Mentor:
Chronicle:	Demeanor:	Cabal:

Attributes

Physical		Social		Mental	
Strength	●OOOO	Charisma	●OOOO	Perception	●OOOO
Dexterity	●OOOO	Manipulation	●OOOO	Intelligence	●OOOO
Stamina	●OOOO	Appearance	●OOOO	Wits	●OOOO

Abilities

Talents		Skills		Knowledges	
Alertness	OOOOO	Drive	OOOOO	Computer	OOOOO
Athletics	OOOOO	Etiquette	OOOOO	Cosmology	OOOOO
Awareness	OOOOO	Firearms	OOOOO	Culture	OOOOO
Brawl	OOOOO	Leadership	OOOOO	Enigmas	OOOOO
Dodge	OOOOO	Meditation	OOOOO	Investigation	OOOOO
Expression	OOOOO	Melee	OOOOO	Law	OOOOO
Intuition	OOOOO	Research	OOOOO	Linguistics	OOOOO
Intimidation	OOOOO	Stealth	OOOOO	Medicine	OOOOO
Streetwise	OOOOO	Survival	OOOOO	Occult	OOOOO
Subterfuge	OOOOO	Technology	OOOOO	Science	OOOOO

Advantages

Spheres

Correspondence	OOOOO	Life	OOOOO	Prime	OOOOO
Entropy	OOOOO	Mind	OOOOO	Spirit	OOOOO
Forces	OOOOO	Matter	OOOOO	Time	OOOOO

Backgrounds

_____ OOOOO
_____ OOOOO
_____ OOOOO
_____ OOOOO
_____ OOOOO

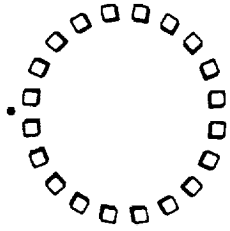
Arete

○○○○○○○○○○

Willpower

○○○○○○○○○○
□□□□□□□□□□

Quintessence



Paradox

Health

Bruised	-0 □
Hurt	-1 □
Injured	-1 □
Wounded	-2 □
Mauled	-2 □
Crippled	-5 □
Incapacitated	□

Experience

Study Points

Combat

Weapon	Difficulty	Damage

Attributes: 7/5/3 Abilities: 13/9/5 Spheres: 5 Backgrounds: 7 Freebie Points: 15 (7/5/4/2/1)

PROGRAMMING

The CLIPS programming language was selected for its relative ease of use, availability, and applicability to the problem solution. The program was originally developed on a networked PC and later moved to an Apollo Domain 4000 in order to run the 600 rules together. The program organization revolves around the characteristics to be developed. One set of questions needed to be developed for the attributes, with a subset of questions for each of the physical, social and mental attributes. Another set of questions had to be developed for abilities, with a subset of questions for determining point values for the various talents, skills and knowledge abilities. Finally, a summary or tally of the resulting character development must be given to the user.

For the attributes, the three areas are prioritized by asking the user which one is most important to them and then which one is secondarily important to them. Then according to the rules, points are assigned to each choice with first, second, and third receiving seven, five, and three points respectively. The user is then asked to work with the first choice and the three divisions under it. The divisions represent sub-areas of each attributes such as physical which contains strength, stamina, and dexterity. Next a strategy was developed for distributing the points to the sub-areas. Since the point amounts are relatively low, one multiple choice question is asked of the user where all three sub-areas are possible answers. For the first choice, the user would receive seven questions, second would receive five, and third three. Since each of the sub-areas would represented in every question, a running total is kept of the number of times a sub-area is picked, and when questioning for that choice is done, that number becomes its level. This strategy is implemented for all three attribute choices.

As an example, the following questions are asked to support point distribution in the social attributes and a summary is given:

```
You have been alloted 7 points for your first choice: social
Please answer the next questions as truthfully as possible.

Which is more important?
<leading> people, <persuading> them, or <looking-good> for them: leading
being <smooth>, <eloquent>, or <alluring>: alluring

Consider these scenarios and type your response.
At a party, you
<1> are generally sociable, talking to all,
<2> grab the center-of-attention, or
<3> are the trend-setter: 3

After a concert,
<1> you talk so well that people think you are in the band,
<2> are asked backstage by the band personally, or
<3> you can talk the bouncers into letting you and all of your
    friends backstage: 2

When going to a job interview, you
<1> make an excellent first impression,
<2> impress them with your manners and attitude, or
<3> convince them how valuable you would be to them: 2

Which activity are you best-suited for?
<1> recruitment for clubs and organizations,
<2> fashion modeling, or
<3> double-talking: 2

Due to your answers to the preceding questions, your points
have been alloted as follows:
Charisma     : 3  "Good: People trust and confide in you."
Manipulation : 2  "Average: Others might believe you."
Appearance   : 5  "Outstanding: First reactions are either that of awe, intense
jealousy or complete solitude."
(You have automatically been given one point in each area)
```

In the example above, the user is presented with three different types of questions: synonyms of the sub-areas names, situations, and best-suited for what-type question. A situation is:

> When going to a job interview, you (1) make an excellent first impression; (2) impress them with your manner and attitude; or (3) convince them how valuable you would be to them.

A choice of number one would be a part to the sub-area of appearance, a choice of two would be charisma, and a choice of three would be manipulation. Once the questions end, the summary comes up with the point totals in each area and a short explanation of what that point level means, as given in the Mage handbook.

For the abilities, the user must again prioritize the three abilities from first to third choice. Points are assigned but in different amounts: first choice receives thirteen, second receives nine, and third five. More points are assigned because of the greater amounts of abilities a character can possess. In this section, the same strategies for point assignment as in the attributes cannot be used because of the larger number of points assigned so a new strategy was needed. Since it is a player's choice whether or not to possess any ability, the expert system gives them that choice. Instead of multiple choice questions, the questions become yes or no responses that describe a type of ability and then gives the choice of having the ability or not. If the ability is not chosen the next ability appears for questioning, but if the ability is wanted, the user types yes and a second part of the question appears to determine the number of points to assign to that ability. Since the user has already selected the ability, one point is automatically assigned in that area and one point is subtracted from the point total. All traits on the character sheets are ranked from zero to five, but since the character has one point in the area, the area need only be ranked from one to five. The second part of an ability question asks the user to rate the character in the ability on a scale of one to five. The number typed becomes the character's level in that ability, and that number is also subtracted from the point total. Questioning continues until all the points are distributed and each ability is covered. At the end of the questioning, a summary appears showing where each point went and what each point level means.

An example of a couple of rules are given showing the code as written by Felicia for the user being asked if the character has some science ability in the knowledges area, and the rule on dealing with a positive response in order to assign the number of points to science. The following page show two rules. For program flow, to go from one question or line of questioning to another, it is required that the previously asserted statement be in the conditions part of the rule. So for each possible outcome, there is a unique asserted statement and one to match it in the appropriate following rule. In the following rule, there are two possible outcomes. If the response is yes, the program asserts (abi-kno-thi-sc TRUE), and if no (abi-kno-thi-g3 TRUE).

```
(defrule abi-kno-thi-q3-ch2
?x <- (abi-kno-thi-q2 TRUE)
?xkno <- (curptsknow ?curptsknow)
?xn <- (numqk ?numqk)
=>
(retract ?x)
(printout t "Number of points left   : "?curptsknow crlf)
(printout t "Number of questions left: "?numqk crlf)
(bind ?numqk (- ?numqk 1))
(retract ?xn)
(assert (numqk ?numqk))
(printout t "Do you have a basic understanding of the different applications of
(printout t "(i.e. physics, chemistry, botany, biology, etc.): ")
(bind ?sci (read))
(while (&& (neq ?sci y) (neq ?sci n))
   (printout "y/n: ")
   (bind ?sci (read)))
(bind ?sci (lowcase ?sci))
(printout t "" crlf)
(if (eq ?sci y)
   then (bind ?ptssci 1)
        (bind ?curptsknow (- ?curptsknow ?ptssci))
        (retract ?xkno)
        (assert (curptsknow ?curptsknow))
        (assert (abi-kno-thi-sci TRUE))
   else (if (eq ?sci n)
           then (retract ?xkno)
                (assert (curptsknow ?curptsknow))
                (bind ?levsci 0)
                (assert (levsci ?levsci))
                (bind ?meansci "")
                (assert (meansci ?meansci))
                (assert (abi-kno-thi-q3 TRUE)))))

(defrule abi-kno-thi-science-pt2-ch2
?x <- (abi-kno-thi-sci TRUE)
?xkno <- (curptsknow ?curptsknow)
=>
(retract ?x)
(printout t "On a scale of 1 to 5, rate your knowledge in science. " crlf)
(printout t "(Use points carefully): ")
(bind ?levsci (read))
(while (&& (neq ?levsci 1) (neq ?levsci 2) (neq ?levsci 3) (neq ?levsci 4) (neq
   (printout t "1 to 5: ")
   (bind ?levsci (read)))
(assert (levsci ?levsci))
(if (eq ?levsci 1)
   then (bind ?meansci "Student: You can make smoke bombs w/ a chemistry set.")
        (assert (meansci ?meansci))
        (bind ?ptssci 0)
        (bind ?curptsknow (- ?curptsknow ?ptssci))
        (retract ?xkno)
        (assert (curptsknow ?curptsknow))
   else (if (eq ?levsci 2)
           then (bind ?meansci "College: You understand the major theories and a
                (assert (meansci ?meansci))
                (bind ?ptssci 1)
                (bind ?curptsknow (- ?curptsknow ?ptssci))
                (retract ?xkno)
                (assert (curptsknow ?curptsknow))
           else (if (eq ?levsci 3)
                   then (bind ?meansci "Masters: You could teach high-school sci
                        (assert (meansci ?meansci))
                        (bind ?ptssin 2)
                        (bind ?curptsknow (- ?curptsknow ?ptssci))
                        (retract ?xkno)
                        (assert (curptsknow ?curptsknow))
                   else (if (eq ?levsci 4)
                           then (bind ?meansci "Doctorate: You might win a Nobel
                                (assert (meansci ?meansci))
                                (bind ?ptssci 3)
                                (bind ?curptsknow (- ?curptsknow ?ptssci))
                                (retract ?xkno)
                                (assert (curptsknow ?curptsknow))
                           else (if (eq ?levsci 5)
                                   then (bind ?meansci "Scholar: Albert Einstein
                                        (assert (meansci ?meansci))
                                        (bind ?ptssci 4)
                                        (bind ?curptsknow (- ?curptsknow ?ptssci
                                        (retract ?xkno)
                                        (assert (curptsknow ?curptsknow)))))))
```

CONCLUSIONS

The subject matter experts actually used the advisor system the day of the ISM Spring Presentation as part of the presentation speech. Our experts agreed the system was effective in cutting the time usually needed for making a character. They generally liked the results given by the system. Although some experts agreed the system worked well, they said they still would prefer the old way with paper and pencil for tradition's sake, but could utilize the advisor as an additional tool to assist in character preparation.

Even though only sixty percent of the entire character sheet for Mage was programmed, the testers and experts agreed the system was efficient and solved the problem of character-building time consumption. The rest of the character sheet could be easily added the existing program. This work shows one potential way for expert systems to move into the entertainment arena, gaming in particular. We expect the basic program framework could be generalized for other role-playing games. A question bank could be developed from which a GM would select questions to be asked of the user in defining the characteristics for use in any role-playing game.

ACKNOWLEDGMENTS

We would like to acknowledge NSF for the funding provided to SwRI, Mrs. Judith Hooper for heading up the Northside Independent School District's ISM program and for being the school liaison for the YES program, the San Antonio role-playing gamers, and Sanjeev Venkatesan and David Lincoln of SwRI.

REFERENCES

Waterman, Donald; A GUIDE TO EXPERT SYSTEMS; Addison-Wesley Publishing Co.; Reading, Massachusetts; 1986; Pages 1-39, 127-176.

Wieck, Stewart; MAGE: THE ASCENSION; White Wolf; 1993; Pages 1-113.

The Computer Aided Aircraft-design Package (CAAP)

By Guy U. Yalif
(617) 973-1015

Abstract

The preliminary design of an aircraft is a complex, labor-intensive, and creative process. Since the 1970's, many computer programs have been written to help automate preliminary airplane design. Time and resource analyses have identified, "a substantial decrease in project duration with the introduction of an automated design capability" (Ref. 1). Proof-of-concept studies have been completed which establish "a foundation for a computer-based airframe design capability" (Ref. 1). Unfortunately, today's design codes exist in many different languages on many, often expensive, hardware platforms. Through the use of a module-based system architecture, the Computer Aided Aircraft-design Package (CAAP) will eventually bring together many of the most useful features of existing programs. Through the use of an expert system, it will add an additional feature that could be described as indispensable to entry level engineers and students: the incorporation of "expert" knowledge into the automated design process.

Introduction

It is widely recognized that good engineers need not only the textbook knowledge learned in school, but also a good "feel" for the designs with which they are working. This "feel" can only be gained with both time and experience. An expert system is an ideal way to codify and capture this "feel". This idea is the key feature of CAAP. With this package, engineers will be able to use the knowledge of their predecessors, as well as learn from it. The potential value of such a program in aiding the engineering professionals as well as the student is great.

The ultimate goal of CAAP is to design a plane in an intelligent way based on user specifications. A rough-sizing configuration is created from user inputs and then analyzed using rule based programming. Throughout the design process, the user is given total access to the CAAP database, which is implemented using object oriented programming. The user can see how variables affect each other, view their present values, and see, create, and arrange rules in a customizable fashion using "Toolbox" files. CAAP exists as a core program with Toolbox files that add functionality to that core, similarly to the popular program "MATLAB". CAAP's core program has been written while its Toolbox files are still in development.

System Overview

Preliminary aircraft design, as described in above, is a multi-faceted problem whose features have driven the choice of software platform used to implement CAAP. This section will detail the features that led to a CLIPS based implementation for CAAP. One aspect of the usefulness of an expert system to the CAAP package has already been discussed.

The design process is a *potentially iterative* procedure. This is best explained with an example. During a hypothetical airplane design, one might re-size the wing five times. On the other hand, it is possible that the engineer will not alter the original fuselage. The possibility of iterative re-design for some components and not for others defines a potentially iterative process. Airplane design is such a process, and it is therefore well modeled by the rule based programming syntax of an expert system.

As other designers have noted, "tremendous amounts of data and information are created and manipulated [during the aircraft design process] to produce numerous parts which are eventually

assembled to a sophisticated system. ... It is becoming clear that a critical issue to effective design is the efficient management of design data" (Ref. 2). The data produced during the design process is voluminous at the very least, but it is not haphazardly arranged. The information needed to design a plane falls into organized patterns. Specifically, a hierarchical structure exists for most components of an airplane. One such component is engine classification, as illustrated in Figure 1. This figure diagrams the hierarchy that is used to describe engines in CAAP. This type of logical hierarchy exists throughout the airplane design process.

The data used during airplane design can be very complicated. Each part of the plane, such as the engine, has its own specifications. Each part also contains subparts, just as the engine has a turbine, compressor, and fuel pumping system. Each of these subparts has its own specifications in addition to sub-subparts. Therefore, design data needs to be arranged in an ordered manner that is readily accessible and understandable to the user. Object Oriented Programming is useful for storing the complex, voluminous, and hierarchically arranged data produced during airplane design. The usefulness of OOP has been recognized elsewhere in the aerospace industry. In a study entitled "Managing Engineering Design Information" (Ref. 2), ten different data storage methods were examined. The conclusion: "The object-oriented data model was found to be a better data modeling method for modeling aerospace vehicle design process" than any of the others studied (Ref. 2).

OOP also facilities the organization of the large number of routines available to aid in aircraft design. Effective routine organization is a desirable quality of any airplane design program. CAAP seeks to accomplish routine organization in two ways. First, routines are grouped into the Toolbox files introduced above. Second, within each Toolbox, different equations are applied to different parts of the airplane as is appropriate. Having the ability to separate the equations according to airplane component aids in the logical organization of the program. Such separation also increases the efficiency of CAAP. For example, it would be a waste of computational time to have the aspect ratio rule searching instances of the FUSELAGE class for possible span and area values. OOP and CLIPS run-time modules allows the programmer to implement such class-specific routine separation.

As discussed above, the order of execution of the routines that analyze an airplane cannot be determined before run-time because of the potentially iterative nature of design. The routines themselves, however, are composed of equations that *do* need to be executed in a predetermined order. For example, the routine that determines the range-payload curve needs to add up the range covered during climb, cruise, and descent over and over again until the desired limiting payloads are reached. This is an ordered process that is best modeled by a procedural programming paradigm.

The desirability of using multiple programming paradigms has been discussed above. Because of these needs, CLIPS was chosen to implement CAAP. CLIPS provides useful and effective rule based, object oriented, and procedural programming paradigms as well as a high level of cross-paradigm interactions. CLIPS is also offered on a wide variety of hardware platforms, ensuring its ability to the student and professional alike.

A Macintosh platform was used to implement these program design goals. The Macintosh was chosen because of the widespread access to Macs, as opposed to the limited access available to more powerful UNIX workstations such as IRIS's or Sun's. Nonetheless, if a user had access to these workstations, CAAP would be fully portable to their environments. CAAP's text based user interface has been written completely in CLIPS. A second, graphically based user interface, however, has been designed strictly for Macintosh use. CAAP has been successfully run on many different Macintosh models, although no porting tests have been performed for other platforms.

It has been recognized that a modular layout "will preserve the ability for independent expansion at a later date" (Ref. 3). This key concept is reflected in CAAP's design. As Kroo and Takai have succinctly stated, "If the founding fathers did not envision that future aircraft might employ maneuver load alleviation, winglets, or three lifting surfaces, it may be more expedient to rewrite the code than to fool the program into thinking of winglets, for example, as oddly shaped stores" (Ref. 4). With a sectioned program, such problems can be quickly alleviated with the addition of another Toolbox or an upgrade to an existing one.

As mentioned above, CAAP is organized into a central program and a variety of Toolboxes which add functionality to the base program. The core program represents all of the code that is necessary to run an expert system which utilizes CAAP's data structures. As a result, this code has the possibility of being recycled in the future. The Toolboxes will add functionality to the core program. If someone wishes to run CAAP at all, they need to possess the core program. If someone also wishes to perform weight sizing of their aircraft, they also need to possess the Weight Toolbox. If a certain Toolbox is not present while a user is designing an airplane with CAAP, the part of the analysis that the Toolbox is responsible for will not be completed. Continuing with the previous example, if the Weight Toolbox is missing, no weights will be assigned to the various components of the plane. Missing Toolboxes could prevent an airplane from being completely designed. Nonetheless, arranging a program in this fashion allows users to customize their personal version of CAAP as they desire. The user will, as a hypothetical example, have the ability to choose a Roskam Weight Toolbox instead of a Raymer Weight Toolbox, if they so desired. In addition, if the user does not want CAAP to perform a certain kind of analysis, the Toolbox based program allows them to disable a segment of code analysis easily. It is worth noting that no time consuming re-compilation is presently necessary to remove a Toolbox from CAAP. A simple menu option allows users to choose which Toolboxes will be loaded at start-up.

Mimicking the real engineer, CAAP's core program will prompt the creation of a rough, initial sizing for an airplane. The code will then analyze this initial configuration of the plane. If the given configuration does not meet one of the user's performance specifications, or if the plane does not pass the appropriate FAR regulations, CAAP will modify a particular part of the plane.

A diagram of the system architecture that will accomplish these tasks is presented in Figure 2. The diagram depicts the interactions between CAAP's run-time modules. Into each of these run-time modules will be loaded routines that perform certain functions in the airplane design process. For example, the Performance Module will contain routines that determine performance estimates. Run-time modules are not to be confused with Toolbox files. Toolbox files are files that contain routines organized in an arbitrary manner chosen by the user. Run-time modules group routines by functionality only.

The CAAP core program is presented around all of the run-time modules in order to emphasize that it is the core code that drives all of the routines within the run-time modules and allows them to perform their allotted analytical tasks. The Initial Sizing Module produces the initial parameters for the first configuration. The rest of the modules then analyze, alter, and re-analyze the subsequent configurations. The final plane will be presented to the user through the routines in the Geometry Graphics Module. The user will then be able to change the proposed solution/configuration, and the process will start over again. This time, however, the "old" solution with the user's modifications will become the "initial configuration".

The Rule Writing Utility

The rules that make up CAAP need not only to perform their prescribed functions, but also to provide variable dependency information to the routine that performs variable assignments for CAAP. As described later in the *Consistency Maintenance* section, if a variable in the system is

calculated, all of the variables that depend on its value must be recalculated. It is useful to maintain a data storage system which can provide CAAP with these variable dependencies. Rules are no longer written in their traditional format. In previous versions of CAAP, when a program developer wanted to add a rule to the expert system, they have had to learn the syntax of rule writing in CLIPS and then how to hard code the rule into the system. This required coding some standard constructs that perform some of the repeated type checking that goes on within CAAP. These constructs were usually very long and "messy", and therefore very time consuming to write.

In CAAP Version 2.0, programmers can add rules to the system by using the Rule Writing Utility (RWU). In order to add a rule, the programmer creates an instance of the class RULE_INFORMATION. They can do this manually or with option 8 of CAAP's main menu. Both methods create objects containing several slots: one set of slots is created listing the input variables and another set is created listing the output variables of the rule that is represented by the instance. The input variables can be restricted to be equal, not equal, less than, greater than, less than or equal to, or greater than or equal to some other value. The text of the calculation that will be executed by the actual rule is also stored in a slot of the instance of RULE_INFORMATION. Therefore, the programmer needs only enter two sets of variables (input and output) and a string representing a calculation (and some housekeeping information). The rest is handled by the RWU.

Before system operation beings, the RWU code creates the class RULE_INFORMATION and the rule "make_rules." "Make_rules" creates expert system rules and places them into the Rule Base based on the existing instances of RULE_INFORMATION. It also adds all of the constraint checking that is necessary for proper CAAP operation. Such a utility could be useful in other Expert Systems that involve the same type of input and output for each rule. As previously mentioned, such code recycling opportunities are an important aspect of CAAP.

Once "make_rules" has fired, each rule is represented in two places: one sense of the rule exists in the Rule Base as an actual expert system rule. Another sense of the rule exists in the Object Oriented Database as an instance of the class RULE_INFORMATION. This second representation of the rule is used by the assignment routine to satisfy its need to know how variables depend on each other. Assignment routine operation and the way the routine uses variable dependency information are described in the *Consistency Maintenance* section.

The present method of double representation is more efficient than what was possible with CLIPS 5.1. Previously, if a programmer wanted to add a rule with five inputs and five outputs, they would have to check that 5 X 5 or twenty-five separate variable dependencies were included in the dependency functions (in addition to the constraint checking information). In Version 2.0, the programmer simply needs to list the five inputs and the five outputs of the rule at the time of RULE_INFORMATION instance creation. This brings CAAP a large step closer towards decreasing the future programmer's work load. Additions to the Rule Base can now be generated more easily and more quickly than before.

The Core Program

CAAP's Core Program has been given the basic abilities needed to design an airplane. At the time of this writing, the Core Program still requires the addition of large amounts of data in the form of Toolbox files in order to be able to design an airplane. Nonetheless, the full functionality of the Core Program has been implemented. The main menu of the package appears in Figure 3. Descriptive English equivalents are used at every point throughout the user interface. A lot of attention was focused on making CAAP user friendly in order not to loose any potential users due to the newness of the program.

The user is presently allowed to create airplanes, modify the specifications the airplane is designed to, and change the airplane's describing variable values. The user may save and load airplanes from disk. The user is also given complete access to the CAAP database. Designers are allowed to look at the variable values that represent an airplane, either individually or in a summary sheet format. They may also look at the dependencies that exist within the CAAP database. This valuable tool would tell the user, for example, that the aspect ratio depends on the wing area and the wing span for its values (reverse dependencies), and that when the wing span is changed, the aspect ratio will have to be recalculated (forward dependencies).

The user is allowed to see a list of all variables which have been defined to CAAP and which are not used in any of the presently loaded rules. This can help designers load the appropriate Toolboxes or add rules where necessary. Users can write rules during run-time and add them to Toolbox files. This feature allows for simple expansion of CAAP by individual users in the future, and, combined with the Toolbox manipulation functions, has allowed CAAP to become a self-modifiable program. With this ability, CAAP can evolve to meet individuals needs as they create and change Toolbox files and the rules that populate them.

The user is given some aesthetic controls over CAAP output. The user can look at CAAP's internal variables representations. Toolboxes can be created, deleted, and loaded during run-time. Users can view which Toolboxes have been loaded into the CAAP database as well as choose which Toolboxes to load each time CAAP starts up. The dynamic use of Toolbox files presents some interesting situations. The files can act as a medium for knowledge exchange in the future. For example, Joe can design a plane with the Toolboxes he has built over time. He can then add Mary's rules to his personal Toolbox, and redesign his plane in order to discover how Mary's know-how can improve his model. Such an interactive exchange of information could be very useful, especially in an teaching environment.

Consistency Maintenance and The Availability of Several Routines to Calculate One Variable

There are several different methods available to estimate almost any of the parameters used in airplane design. Different sources will quote different methods, each with its own result. A consistent method for routine execution is needed. When there is more than one equation or routine available to calculate a given parameter, CAAP will select the most advanced one for which all of the input variables have been determined. For example, the airplane drag coefficient can be calculated using equation (1) or with a drag build-up.

$$C_{L_\alpha} = C_{l_\alpha} \frac{A}{A + [2(A+4)/(A+2)]} \qquad (1)$$

C_{L_α} = lift curve slope for a finite lifting surface

C_{l_α} = section lift curve slope

A = aspect ratio

If the components of the plane have been defined, the latter, more advanced drag estimation method will be used. If the components have not yet been defined, the former, simpler method will be used. Importantly, once the components have been defined, the LHS of the drag build-up rule will be satisfied and CAAP will *recalculate* the drag based on the more advanced drag build-up. All calculations are based on the most advanced routine available, due to the rule based programming implementation chosen for CAAP.

Rule "advancedness" will be represented by a priority associated with each rule. This priority is stored in a "rulepriority" function in CAAP's core program. It is presently used to ensure that the

Inference Engine sees more "advanced" rules more quickly than it sees more primitive rules. Rulepriority is used by the RWU to set rule saliences during system initialization. This procedure improves program efficiency by decreasing the likelihood that a particular variable will be calculated many times by successively more advanced routines when a very advanced one could have done the job originally. Rule prioritization also allows the user to be confident that the crude initial estimates used in the Initial Sizing Toolbox will not be used in the final configuration. As soon as the airplane begins to take shape, the Initial Sizing Toolbox's estimates will be replaced with more advanced values[1].

If two methods are similarly "advanced", one method will be chosen over the other arbitrarily, but not randomly. If the designer has a preference as to which method is used by CAAP, he or she can specify this to the package. The "rulepriority" function alleviates the need for addressing the situation when two expert recommendations agree. Either they will have different priorities, or their location on the agenda will determine which is fired.

Consistency Maintenance and Parameter Modifications

During the design process, configurations are created and analyzed. If the analysis shows a given configuration to be inadequate in some way, the rules within the Expert run-time module will modify one of the design parameters of the given configuration, in effect creating a new configuration. Until the effects of this single modified parameter have been propagated throughout the airplane, the configuration will be inconsistent. In another scenario, an advanced routine might recalculate a design parameter previously calculated by a more primitive routine. Again, until the effects of this change have been propagated throughout the system, an inconsistent configuration will exist. The solution to this problem follows.

Consistency maintenance will be accomplished in two ways. When a rule within the Expert run-time module modifies an airplane design parameter, it will have to do so in a "responsible" manner. For example, suppose the Expert rule, for an "expert" reason, decides that the aspect ratio of the wing needs to be changed. If it simply changes the aspect ratio, the span and/or wing area will be inconsistent. Therefore, the Expert rule will have to *also* change the span or the wing area. The rule could, for example, adjust the aspect ratio while keeping the wing area constant. In other words, the Expert rule will have to look at the input variables that determine the value of the design parameter and modify them so that they are consistent with the new value of the changed variable.

The second consistency maintenance procedure will be based on computational paths. Figure 4 presents a diagram of a hypothetical set of computational paths. Each box on the diagram represents a variable. The directed connections represent the dependency of a variable on the value of other variables.

Suppose that the variable in the shaded box has just been redefined, perhaps by a rule from the Expert run-time module or by an advanced estimation routine. The value of every box "downstream" of the shaded box is now inconsistent. The "downstream" variables are represented by the presence of an "X" in the variable box. The "downstream" variables need to be recalculated as if they had never been determined in the first place. Each rule will have access to the list of variables which depend on the variable in the shaded box (i.e. X_1 in this example). This list is stored in instances of the RULE_INFORMATION class, introduced in the *Rule Writing Utility*

[1]This does not necessarily have to occur, if one is not careful. It would be possible for the airplane to be presented as a final product without enough of it having been calculated to replace the Initial Sizing Toolbox's estimates. This would be an absurd situation, and it would result in problems. CAAP will not present a plane to the user unless a minimum set of parameters have been calculated to a sufficient level of "advancedness". This way, no Initial Sizing Toolbox estimates will make their way to the user.

section. The rule will erase, or undefine[2], all of the variables that depend on the changed variable (i.e. it will undefine X_1). The Toolbox will then undefine all of the variables that depended on those variables, and so on until the there are no more dependent variables to undefine (i.e. X_2, X_3, ... , X_6). This systematic undefining is called the "downstream" erasure procedure. It has been coded as part of the "assign" routine that is used for all variable assignments. Every rule must use the "assign" routine. After a "downstream" erasure, the other rules in CAAP will automatically recalculate the undefined "downstream" variables. This will occur since the LHS of CAAP rules is satisfied when the input variables for the rule are undefined.

A problem with the method of consistency maintenance presented in Figure 4 will arise if any loops exist within the computational paths. A discussion of this problem is beyond the scope of this paper, and the problem has only been partially solved. A full solution to the "Loop Problem" is one of the major remaining issues facing CAAP.

Practical Limitations

The future of CAAP will focus on three different areas: the core program, the Toolboxes, and the user interface. The essentials of the core program have been entirely written. Some extra functionality has also been added to the program. Nonetheless, there is always room for improvement and CAAP is by no means complete. Among the pieces of code not yet written is a numerical optimizer. Such code could provide CAAP with a way to make "expert" recommendations when no rules from the Expert run-time module apply to a given configuration. If no rules exist to help, CAAP could turn to numerical optimization methods in order to determine what changes to make to a configuration in order to make it meet all user and FAR requirements. A simultaneous equation solver could significantly facilitate solving the airplane design problem.

The Toolbox files need to receive a significant amount of data. Proof of study Toolbox files have been implemented and successfully tested, but there remains a lot of data to input in order to fully design an airplane. The graphical user interface ran into difficulties associated with system level Macintosh programming. Finding an alternative to friendly user interactions will be a priority for CAAP in the future.

The first category of plane that CAAP should be able to completely design will be the light, general aviation, non-acrobatic, single engine aircraft. The graphics for displaying the airplane are next on the implementation list. Eventually, trend studies and increased user involvement in the design process could be added. For example, if the user wished CAAP to produce several final designs instead of one, this could be done. If the user wished to watch CAAP fire one rule at a time, this could be done. A utility could be added to allow users to see which rules are firing at any given time. This would provide the user with a better "feel" for how the package is going about designing their airplane.

Conclusion

A firm theoretical foundation has been developed for CAAP. The problem of designing an airplane has been laid out and implemented using rule based, object oriented, and procedural programming paradigms. Rule based programming enables CAAP to capture expert knowledge and to mimic the *potentially iterative* nature of preliminary airplane design. Object oriented programming handles the voluminous, complex, and hierarchically arranged data produced during airplane design.

---

[2]CLIPS 6.0 no longer supports undefined slot values. It is necessary to have such reserved values for airplane variables that may take on a range of values. In order to satisfy the LHS's of any of the rules, the LHS's must contain tests for variables to see if they have not yet been calculated, that is that they are undefined. A typical undefined value is -1e-30 for a floating point variable.

Procedural programming is used to implement the actual analysis routines necessary for engineering design. CAAP has realized core program implementation and proof-of-concept Toolbox file creation and test. CAAP can begin designing airplanes and awaits the addition of more data in order to be able to complete the design process. CAAP is still in the developmental phase.

Figures

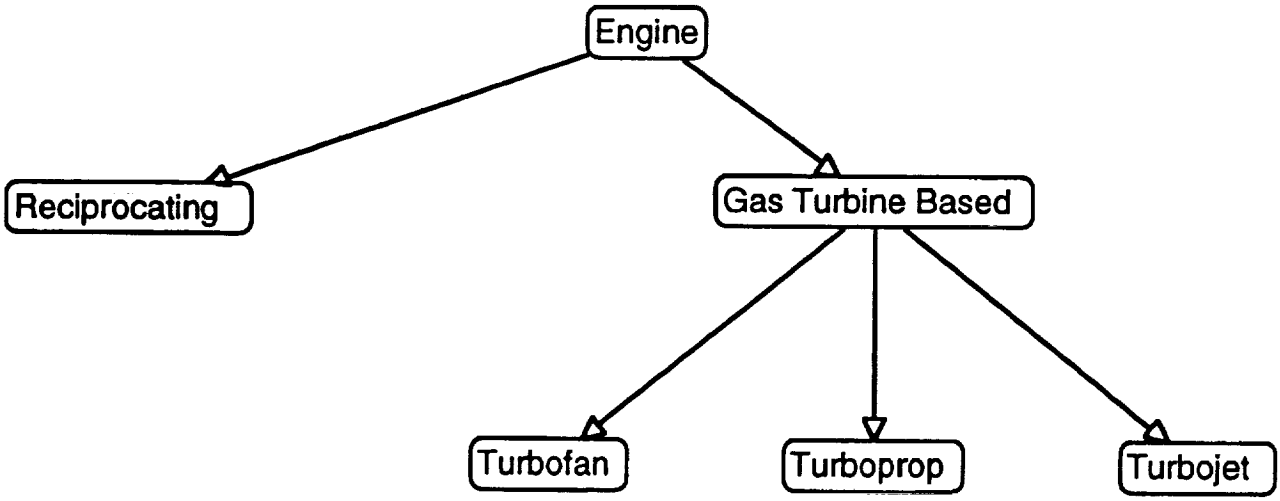


Figure 1 - Engine Classifications in CAAP

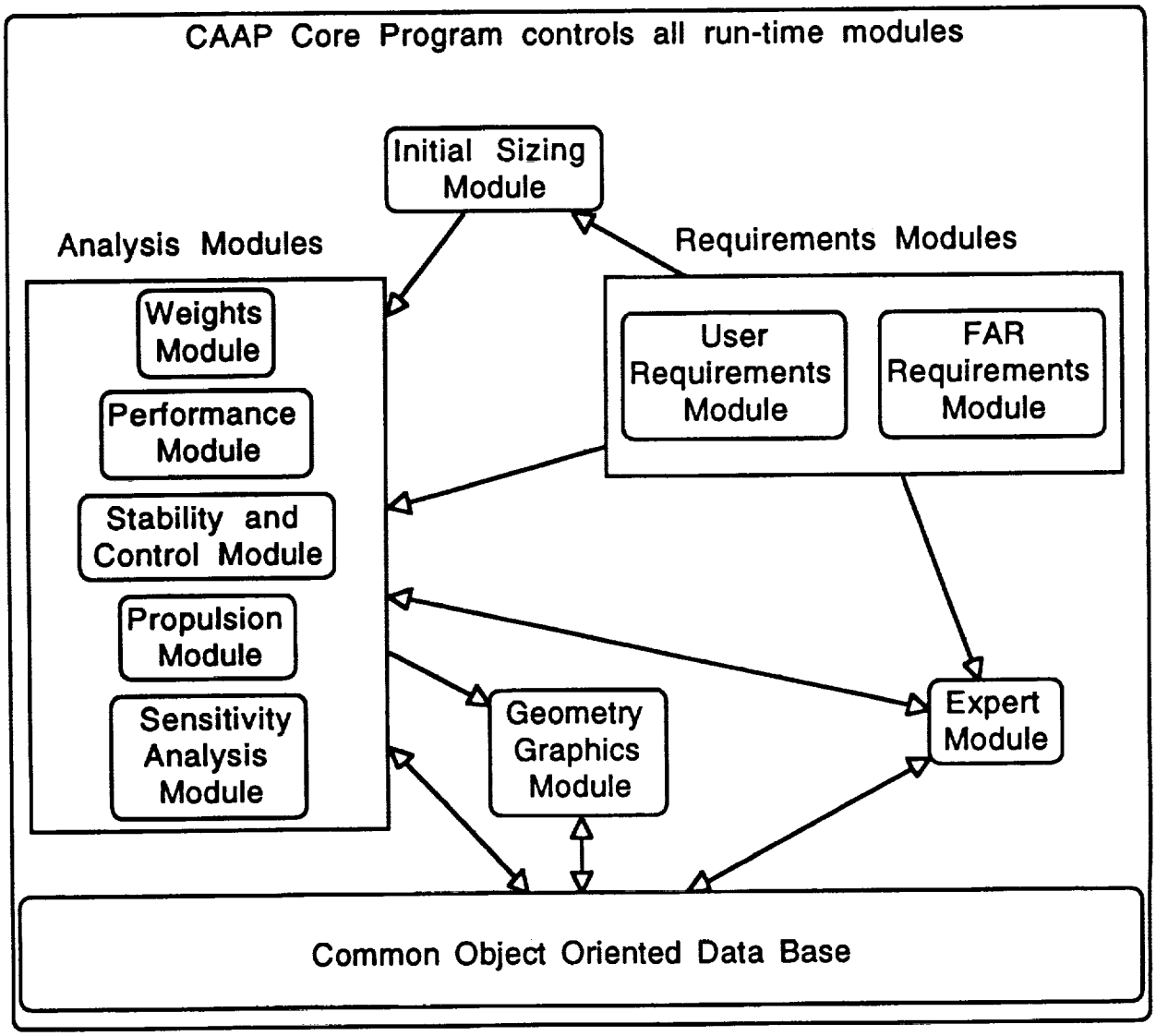


Figure 2 - System Architecture

Main Menu
---
1) Create an airplane
2) Modify requirements or plane
3) Analyze airplanes
4) Save an airplane to disk
5) Load an airplane from disk
6) Look at an airplane
7) Look at slot dependencies
8) Add a rule to the database
9) Look at internal variable representation
10) Manipulate Toolboxes
11) Miscellaneous
12) Quit

Please choose an option (0 to reprint menu)>

Figure 3 - CAAP Main Menu

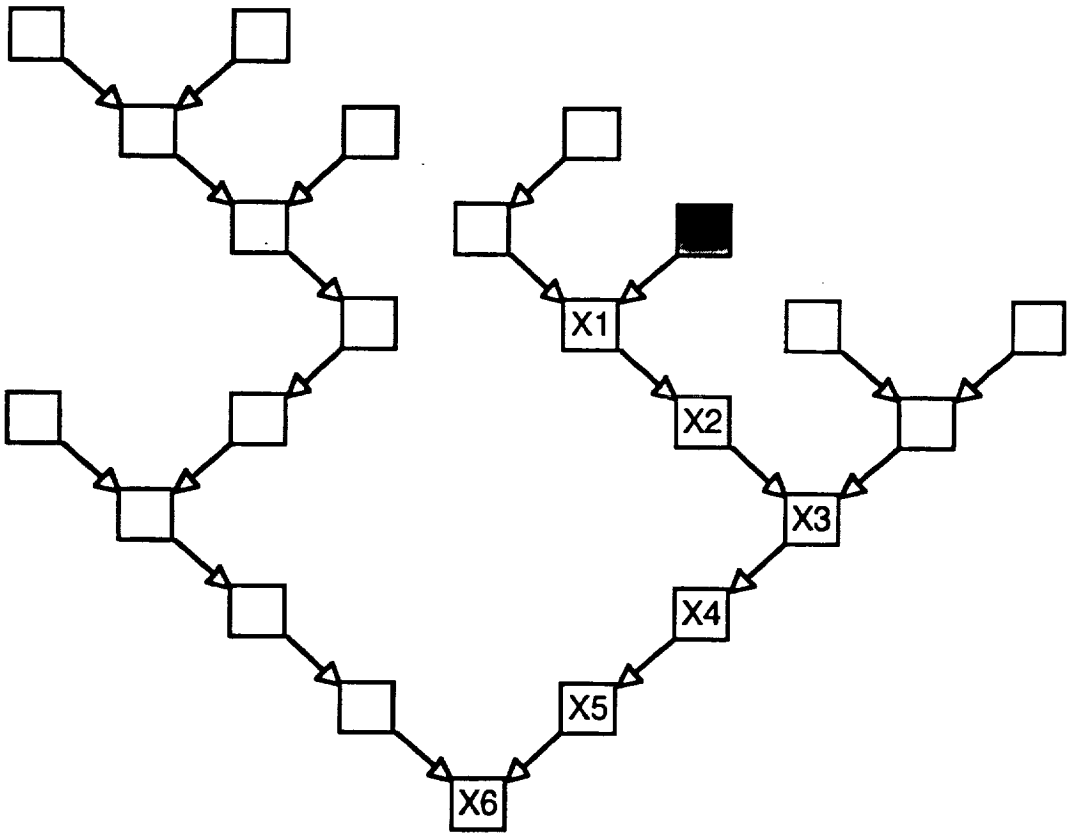


Figure 4 - Consistency Maintenance Example

References

1. Newman, D., and K. Stanzione. Aircraft Configuration Design Code Proof-Of-Concept: Design of the Crewstation Subsystem. Proc. of the AIAA Aircraft Design Systems and Operations Meeting. 23-25 Sept. 1991. Baltimore: AIAA paper No. 91-3097, 1991.

2. Fulton, R. E., and Yeh Chao-pin. Managing Engineering Design Information. Proc. of the AIAA/AHS/ASEE Aircraft Design, Systems and Operations Conference. 7-9 Sept. 1988. Atlanta: AIAA paper No. 88-4452, 1988.

3. Roskam, Jan, and Seyyed Malaek. "Automated Aircraft Configuration Design and Analysis." SAE Technical Paper Series No. 891072 (1989): General Aviation Aircraft Meeting & Exposition (Wichita, KS), 1989.

4. Kroo, I., and M. Takai. A Quasi-Procedural, Knowledge-Based System for Aircraft Design. Proc. of the AIAA/AHS/ASEE Aircraft Design, Systems and Operations Meeting. 7-9 Sept. 1988. Atlanta: AIAA paper No. 88-4428, 1988.

omit

Session 6B: Prototyping and Rule Generation/Revision Extensions

Session Chair: Bebe Ly

RULE BASED DESIGN OF CONCEPTUAL MODELS FOR FORMATIVE EVALUATION

Loretta A. Moore[+], Kai Chang[+], Joseph P. Hale[*], Terri Bester[+], Thomas Rix[+], and Yaowen Wang[+]

[+]Computer Science and Engineering
Auburn University
Auburn, AL 36849
(205) 844 - 6330
moore@eng.auburn.edu

[*]Mission Operations Laboratory
NASA Marshall Space Flight Center
MSFC, AL 35812
(205) 544-2193
joe.hale@msfc.nasa.gov

ABSTRACT

A Human-Computer Interface (HCI) Prototyping Environment with embedded evaluation capability has been investigated. This environment will be valuable in developing and refining HCI standards and evaluating program/project interface development, especially Space Station Freedom on-board displays for payload operations. This environment, which allows for rapid prototyping and evaluation of graphical interfaces, includes the following four components: (1) a HCI development tool, (2) a low fidelity simulator development tool, (3) a dynamic, interactive interface between the HCI and the simulator, and (4) an embedded evaluator that evaluates the adequacy of a HCI based on a user's performance. The embedded evaluation tool collects data while the user is interacting with the system and evaluates the adequacy of an interface based on a user's performance. This paper describes the design of conceptual models for the embedded evaluation system using a rule-based approach.

INTRODUCTION

Formative evaluation is conducted through usability studies. Given a functional prototype and tasks that can be accomplished on that prototype, the designer observes how users interact with the prototype to accomplish those tasks in order to identify improvements for the next design iteration. Evaluation of the interaction is measured in terms of specific parameters including: time to learn to use the system, speed of task performance, rates and types of errors made by users, retention over time, and subjective satisfaction [4]. Analysis of this information will assist in redesign of the system.

The conceptual model of a designer is a description of the system and how the user should interact with it in terms of completing a set of tasks [2]. The user's mental model is a model formed by the user of how the system works, and it guides the user's actions [1]. Most interaction problems occur when the user has an inaccurate model of the system or when the user's model of a system does not correspond with the designer's conceptual model of the system. The evaluation approach which will be discussed in this paper evaluates the user's mental model of the system against the designer's conceptual model.

A rule-based evaluation approach, implemented using CLIPS, is used to develop the conceptual model. The model outlines the specific actions that the user must take in order to complete a task. Evaluation criteria which are embedded in the rules include the existence of certain actions, the sequencing of actions, and the time in which actions should be completed. Throughout the evaluation process, user actions are continuously associated with a set of possibly changing goals. Once a goal has been identified, the user's action in response to that goal are evaluated to determine if a user has performed a task correctly. Tasks may be performed at three levels: expert, intermediate, and novice.

This research is supported in part by the Mission Operations Laboratory, NASA, Marshall Space Flight Center, MSFC, AL 35812 under Contract NAS8-39131, Delivery Order No. 25. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressly or implied, of NASA.

The dynamic relationship between the evaluation tool and the user environment allows the simulation director to constantly introduce new goals that need to be responded to. This paper will discuss the approach of rule-based conceptual modeling and will provide an example of how this approach is used in the evaluation of a graphical interface of an automobile.

ARCHITECTURE OF THE HCI PROTOTYPING ENVIRONMENT

The Human-Computer Interface Prototyping Environment with Embedded Evaluation capability is designed to allow a developer to create a rapid prototype of a system and to specify correct procedures for operating the system [3]. The first component of the architecture is the Graphical User Interface (GUI) development tool. This tool allows the designer to graphically create the interface of the system and specify a data source for each object within the display. The simulator tool provides the capability to create a low-fidelity simulation of the system to drive the interface. The embedded evaluation tool allows the designer to specify which actions need to be taken to complete a task, what actions should be taken in response to certain events (e.g., malfunctions), and the time frames in which these actions should be taken. Each of these components is a separate process which communicates with its peers through the network server. Figure 1 shows the architecture of the HCI Prototyping Environment.

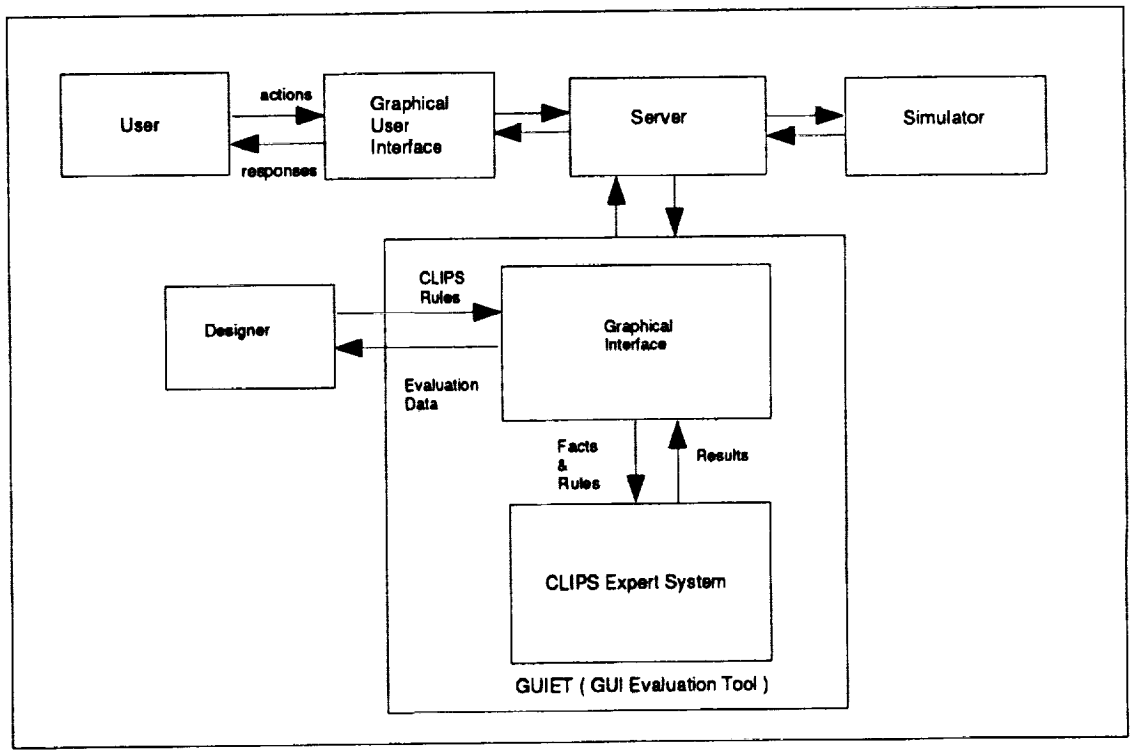


Figure 1 - Architecture of the HCI Prototyping Environment

During execution of the system, the interface objects send and receive data and commands to the simulator by way of the data server and the simulator provides realistic feedback to the interface based on user inputs. The server sends the embedded evaluation tool the actions which the user has taken, all events and activities which have occurred, and the times associated with these items. The embedded evaluation tool analyzes the actions which have been performed by the user, that is, the user's model of the system, against the predefined conceptual model of the designer. The system identifies which tasks were completed correctly, or not, and provides data to the designer as to the points in the interaction in which the user's model of the system did not correspond to the designer's conceptual model of the system.

In order to evaluate the architecture, an automobile system was prototyped in the environment. An automobile was chosen because it has sufficient complexity and subsystems' interdependencies to provide a moderate level of operational workload. Further, potential subjects in the empirical studies would have a working understanding of an automobile's functionality, thus minimizing pre-experiment training requirements. The conceptual model which will be described in this paper is that for the automobile prototype. Before describing the rule based conceptual model, we will first discuss changes made to integrate CLIPS into the HCI Prototyping Environment.

THE GRAPHICAL USER INTERFACE EVALUATION TOOL (GUIET)

The goal of GUIET is to provide for dynamic evaluation of user actions within the HCI Prototyping Environment. Using GUIET, the process of formative evaluation has more flexibility and takes less time for analysis. The main feature is that the evaluation of most of the participant's actions are automated. The evaluation is performed at runtime by an expert system. The knowledge base of the system contains the designer's conceptual model, of how he/she thinks the user should interact with the prototyped system. Because the knowledge base is not hard coded into the application, it can be dynamically changed according to the needs of the evaluator. This provides the flexibility to evaluate different interfaces with the same evaluation criteria or one interface with different evaluation criteria. This design saves time because the data is automatically collected and analyzed based on the rule based conceptual model. If a new interface is prototyped, the only change that needs to be made with GUIET is changing the knowledge base.

In addition to the rule-based modeling design, GUIET provides a graphical interface and communication capabilities to CLIPS. In order to be integrated with the existing architecture, GUIET needs to receive information from both the interface and the simulator. The server sends the messages that are passed between the GUI tool and the simulator tool to GUIET. This is done using datagram sockets for the interprocess communication. The messages are in the form:

(newfact 193.0 I>S Message SetVariable gear 2)

where newfact is a string used for pattern matching for CLIPS rules, 193.0 is the time stamp, I>S states that communication is from the interface to the simulator, Message is the type of communication, SetVariable represents that the value of a variable is being set by the user, gear is the variable name, and 2 is the variable value.

Although this is the natural way to assert facts into CLIPS, the state for the car is stored not as a set of facts, but as one fact with many attribute/value pairs. Before the new fact is evaluated it is translated into the form of a fact. This translation is done by a set of translation rules. An example of a translation rule is:

```
; GEARS
:
(defrule trans_gear
        ?newfact <- (newfact ?time ?direction ?type ?action gear ?value)
        ?state <- (car_state)
=>
        (modify ?state (gear ?value))
        (retract ?new_fact)
        (bind ?*current_tag* (+ ?*current_tag* 1))
        (assert (action (type gear) (value ?value) (time ?time)
                        (clock_tag ?*current_tag*) ))
```

Another set of rules perform the evaluation of the user's actions, which are described in the next section. The last section describes a graphical interface which has been created for CLIPS.

CONCEPTUAL MODEL FOR THE AUTOMOBILE PROTOTYPE

The tasks which the user are asked to perform with the prototype can be divided into two categories: driving the car (i.e., using the controls) and responding to events (e.g., environmental and maintenance). The tasks measured include:

- Starting the car
- Driving forward (including changing gears)
- Driving backward
- Turning
- Stopping (at stop signs, lights, etc.)
- Parking the car
- Increasing and decreasing speed [Responding to speed limit changes]
- Driving uphill and downhill [Responding to hill events]
- Performing maintenance [Responding to maintenance events]
- Responding to environmental conditions

The events which can occur while the user is driving include environmental condition events (e.g., rain, snow, fog, and clear weather), time of day events (e.g., day and night), terrain changes (uphill and downhill), speed limit changes, and maintenance problems (e.g., gas, oil, battery, alternator, and engine). In addition to the events, the participant is given a set of instructions that must be followed. These are in the form of driving directions (e.g., drive 5 miles north and park the car).

Driving the car consists of manipulating graphical objects on the screen. For each of the tasks described above, the designer has determined a set of correct actions that must be made to complete the task. For example, the actions which must be taken for starting the car include:

1. Lock the seatbelt
2. Release emergency brake
3. Depress the brake
4. Depress the clutch
5. Put the gear in neutral
6. Turn the key on

Task correctness is evaluated based mainly on three evaluation criteria: the existence of certain actions, the sequencing of actions, and the time associated with the completions of the actions or task. An integer clock counter is used to indicate the action or event sequence. In the beginning of evaluation, the clock is reset to zero. Every subsequent action taken by the driver would increment the clock by one. Action sequence is important for many driving maneuvers. For example, clutch must be engaged before shifting gears. The evaluation process evaluates the correctness and effectiveness of a driver's interactions with the graphical user interface. User performance can be classified into three levels for most tasks - expert, intermediate, and novice. There may also be no response to a task. A counter is designated for each performance level. Every time a sequence of user actions is classified at a particular level, the associated counter will be incremented by one. The purpose of the evaluation is not to classify or evaluate users, but to evaluate the interface. The classification of users into categories is done to identify the level at which the users are interacting with the system. The goal is to have most if not all interactions at what the designer would consider the expert level. If users are not interacting at this level, it is the interface which must be enhanced to improve user performance.

In the CLIPS implementation a fact template is used to represent the car states and driving scenarios. The following shows the template for car-states:

```
(deftemplate car_states "Variables used in the car interface"
        (slot turnsignal)
        (slot brake)
        (slot emer_brake_on)
        (slot clutch)
        (slot key)
        (slot gear)
        (slot throttle)
        (slot speed)
        (slot seatbelt)
        (slot wipers)
        (slot lights)
        (slot fog_lights)
        (slot oil)
        (slot gasoline)
        (slot engine_temp)
        (slot battery)
        (slot alternator_ok)
        (slot rpm)
        (slot terrain)
        (slot day)
        (slot weather)
        (slot speed_limit)
) ;car template
```

Associated with each action taken by the user, a template is defined as:

```
(deftemplate action "Action taken by the user on the interface"
        (slot type)                     ;action type
        (slot value)                    ;action value
        (slot time)                     ;action time
        (slot clock_tag)                ;action sequence
) ;action template
```

An evaluation rule is designed for each performance level. After a sequence of actions is completed, it will be evaluated based on the rules for the three performance levels. However, only one of the rules would succeed. The rules are organized in a way that the expert level would be tried first, then the intermediate level, and then the novice level. Once a rule has been successfully fired, this sequence of actions will be discarded. The prioritization of these rule is achieved through the *salience* values of CLIPS.

The following examples describe a portion of the evaluation process for *starting the engine.*

```
; Rules for Starting the Engine
(defrule expert_start_engine "Expert level starting engine"
        ; salience value highest among the four performance evaluation rules
```

```
        (declare (salience 3))

        ; This part of the rule ensures that all of the actions have occurred.
        ; the first action putting the seatbelt on
        ?f1 <- (action (type seatbelt) (value 1) (clock_tag ?t))
        ; next action is about the emergency brake
        ?f2 <- (action (type emer_brake_on) (value 0) (clock_tag ?t1))
        ; next action is brake on
        ?f3 <- (action (type brake) (value 1) (clock_tag ?t2))
        ; next action is clutch depressed
        ?f4 <- (action (type clutch) (value 1) (clock_tag ?t3))
        ; next action is gear neutral
        ?f5 <- (action (type gear) (value 0) (clock_tag ?t4))
        ; next action is key on
        ?f6 <- (action (type key) (value 1) (clock_tag ?t5))
        ?f7 <- (noresponse start_engine)

        ; To ensure that the actions occurred in the proper order the following
        ; test is performed. The proper sequence is lock seatbelt, release
        ; emergency brake, depress brake, depress clutch, select neutral gear,
        ; turn key, release brakes and release clutch.

        (test (= (+ ?t 1) ?t1))
        (test (= (+ ?t 2) ?t2))
        (test (= (+ ?t 3) ?t3))
        (test (= (+ ?t 4) ?t4))
        (test (= (+ ?t 5) ?t5))
=>
        ; The following assertion aids in keeping track of whether a response
        ; was made or not.
        (assert (response start_engine))

        ; effects of actions considered, therefore removed from the fact base
        (retract ?f1 ?f2 ?f3 ?f4 ?f5 ?f6 ?f7)

        ; increment expert level count
        (bind ?*expert_count* (+ ?*expert_count* 1))

        (printout evaluation "TASK: starting the engine" crlf)
        (printout evaluation "TIME:  " (- (integer (time)) ?*start_time*) " seconds" crlf)
        (printout evaluation "# ERRORS: 0" crlf crlf)

) ; expert_start_engine

(defrule intermediate_start_engine "Intermediate level starting engine"

        ; salience value smaller than expert level, but higher than novice level
        (declare (salience 2))
```

```
        ; This part of the rule ensures that all of the actions have occurred.

        ; the first action putting the seatbelt on
        ?f1 <- (action (type seatbelt) (value 1) (clock_tag ?t))
        ; next action is about the emergency brake
        ?f2 <- (action (type emer_brake_on) (value 0) (clock_tag ?t1))
        ; next action is brake on
        ?f3 <- (action (type brake) (value 1) (clock_tag ?t2))
        ; next action is clutch depressed
        ?f4 <- (action (type clutch) (value 1) (clock_tag ?t3))
        ; next action is gear neutral
        ?f5 <- (action (type gear) (value 0) (clock_tag ?t4))
        ; next action is key on
        ?f6 <- (action (type key) (value 1) (clock_tag ?t5))
        ?f7 <- (noresponse start_engine)

        ; The following test is to see what sequence the events occurred in.
        ; The proper sequence is: key turned after the clutch is off, clutch off
        ; after the brake is on, and seatbelt is on before the key is turned.
        ; Additional actions may be done in between the needed actions.
        ; For example, turning on the fog lights.

        (test (> ?t1 ?t))
        (test (> ?t2 ?t1))
        (test (> ?t3 ?t2))
        (test (> ?t4 ?t3))
        (test (> ?t5 ?t4))
=>
        ; The following assertion aids in keeping track of whether a response
        ; was made or not
        (assert (response start_engine))

        ; effects of actions considered, therefore removed from the fact base
        (retract ?f1 ?f2 ?f3 ?f4 ?f5 ?f6 ?f7)

        ; increment intermediate level count
        (bind ?*intermediate_count* (+ ?*intermediate_count* 1))

        (printout evaluation "TASK: starting the engine" crlf)
        (printout evaluation "TIME:  " (- (integer (time)) ?*start_time*) " seconds" crlf)
        (printout evaluation "# ERRORS: 1 or more" crlf)
        (printout evaluation "EXPLANATION: Extra events occurred in the sequence." crlf )

) ;intermediate_start_engine

(defrule novice_start_engine "Novice level starting engine"

        ; salience value smaller than intermediate level, but higher than no response level
        (declare (salience 1))
```

```
        ; This part of the rule ensures that all of the actions have occurred.

        ; the first action putting the seatbelt on
        ?f1 <- (action (type seatbelt) (value 1) (clock_tag ?t))
        ; next action is about the emergency brake
        ?f2 <- (action (type emer_brake_on) (value 0) (clock_tag ?t1))
        ; next action is brake on
        ?f3 <- (action (type brake) (value 1) (clock_tag ?t2))
        ; next action is clutch depressed
        ?f4 <- (action (type clutch) (value 1) (clock_tag ?t3))
        ; next action is gear neutral
        ?f5 <- (action (type gear) (value 0) (clock_tag ?t4))
        ; next action is key on
        ?f6 <- (action (type key) (value 1) (clock_tag ?t5))
        ?f7 <- (noresponse start_engine)

        ; The events do not have to occur in any particular order. They just
        ; all have to occur.
=>
        ; The following assertion aids in keeping track of whether a response
        ; was made or not
        (assert (response start_engine))

        ; effects of actions considered, therefore removed from the fact base
        (retract ?f1 ?f2 ?f3 ?f4 ?f5 ?f6 ?f7)

        ; increment novice level count
        (bind ?*novice_count* (+ ?*novice_count* 1))

        (printout evaluation "TASK: starting the engine" crlf)
        (printout evaluation "TIME:  " (- (integer (time)) ?*start_time*) " seconds" crlf)
        (printout evaluation "# ERRORS: 1 or more" crlf)
        (printout evaluation "EXPLANATION: Events occurred out of sequence." crlf)

) ;novice_start_engine

(defrule no_response_start_engine "No response to starting engine"
        ; No salience value is assigned therefore it has the default value of 0

        ; Check to see if there was an attempt to start the engine
        (noresponse start_engine)

        ; Check to see if the time limit has exceeded for starting the engine
        (test (>  (integer (time)) (+ ?*timeout* ?*start_time*)))
=>
        ; increment no response count
        (bind ?*no_response_count* (+ ?*no_response_count* 1))

        (printout evaluation "No response to starting the engine" crlf)
```

```
        (printout evaluation "TIME:  " (- (integer (time)) ?*start_time*) "seconds" crlf crlf)

    ) ;no_response_start_engine
```

Rules for different tasks may contain different evaluation criteria. It depends on the designer's conceptual model of how he/she feels the task needs to be completed.

GRAPHICAL INTERFACE FOR CLIPS

A graphical interface was created for CLIPS using the Motif toolkit. The purpose of this interface is to provide the human evaluator a graphical means by which to use CLIPS. The main window of GUIET is composed of two areas. The top area is used for CLIPS standard input and output, and the bottom area displays any error or warning messages from CLIPS. These areas receive their input from a series of CLIPS I/O router functions. The menu bar for this window provides the following options: System, Network, CLIPS, Rules, and Facts. The system pulldown provides functions for quitting the application and for bringing up a window for entering participant information. The network pulldown allows the evaluator to send a selection of messages to the server (e.g., connect and disconnect). The CLIPS pulldown supports functions that affect the entire expert system, such as loading the evaluation rule base, resetting the expert system to its initial settings, running, clearing, or accepting input. Debugging functions that relate to the rules, such as watching/unwatching the rules fire or viewing/editing existing rules, can be accessed through the rules pulldown. Debugging functions that relate to the facts in the expert system are provided under the facts pulldown.

The user information window accessed under the system pulldown allows the evaluator to enter data relevant to the current participant and system being evaluated. The top section of this window displays the expert system's evaluations. There is an I/O router which handles the display for the evaluation window and is accessed through the printout statement within the rules. The bottom area is a text area in which the evaluator can enter comments or observations about the participant's session. The window provides options for saving, printing, and cancelling information.

CONCLUSION

The rule-based design of conceptual models enables the iterative process of design and evaluation to proceed more efficiently. A designer can specify user performance criteria prior to evaluation, and the system can automatically evaluate the human computer interaction based on the criteria previously specified. In order to evaluate the system which has been designed, a study is being planned which will evaluate user performance (using the rule based system developed) using a good interface and a bad interface. The hypothesis is that the good interface will produce more user responses at the expert level, and the bad interface will produce less acceptable responses.

REFERENCES

1. Eberts, Ray, *User Interface Design*, Prentice Hall, Englewood Cliffs, New Jersey, 1994.

2. Mayhew, Deborah, *Principles and Guidelines in Software User Interface Design*, Prentice Halll, Englewood Cliffs, New Jersey, 1992.

3. Moore, Loretta, "Assessment of a Human Computer Interface Prototyping Environment," Final Report, Delivery Order No. 16, Basic NASA Contract No. NAS8-39131, NASA Marshall Space Fight Center, Huntsville, AL, 1993.

4. Shneiderman, Ben, *Designing the User Interface: Strategies for Effective Human-Computer Interaction*, Second Edition, Addison-Wesley, Reading, Massachusetts, 1992.

S/4-61
34097
P.8

N95-19761

AUTOMATED RULE-BASE CREATION via CLIPS-INDUCE

Patrick M. Murphy

Department of Information & Computer Science
University of California, Irvine, CA 92717
pmurphy@ics.uci.edu
(714) 725-2111

Abstract

Many CLIPS rule-bases contain one or more rule groups that perform classification. In this paper we describe CLIPS-Induce, an automated system for the creation of a CLIPS classification rule-base from a set of test cases. CLIPS-Induce consists of two components, a decision tree induction component and a CLIPS production extraction component. ID3 [1], a popular decision tree induction algorithm, is used to induce a decision tree from the test cases. CLIPS production extraction is accomplished through a top-down traversal of the decision tree. Nodes of the tree are used to construct query rules, and branches of the tree are used to construct classification rules. The learned CLIPS productions may easily be incorporated into a large CLIPS system that perform tasks such as accessing a database or displaying information.

INTRODUCTION

Many CLIPS rule-bases contain one or more rule groups that perform classification. In this paper we describe CLIPS-Induce, an automated system for the creation of a CLIPS classification rule-base from a set of test cases. The rule-base created by CLIPS-Induce consists of two sets of rules, a set of user query rules to ask the user for any missing information necessary to perform classification, and a set of classification rules that is used to make the classification.

In the remainder of the paper, a detailed description of CLIPS-Induce and ID3 will be presented, followed by an analysis of CLIPS-Induce and list of potential extensions.

DESCRIPTION

In this section a description of CLIPS-Induce will be given, along with an example of its usage on a real-world problem, the Space Shuttle Landing Control problem. The goal of this classification problem is to determine whether the space shuttle should be landed manually or automatically.

CLIPS-Induce takes as input a set of test cases and returns two sets of CLIPS rules that perform user querying and classification. Each case is described in terms of a set of feature-value pairs. The same set of features is used for each case. An example case for the shuttle problem is given in Table I.

Table I: Example case from shuttle problem.

- Landing = manual
- Stability = stab
- Error = mm
- Sign = nn
- Wind = tail
- Magnitude = OutOfRange
- Visibility = yes

One feature is identified as the feature to be predicted given the values of the other features. For the shuttle problem, the feature *Landing* is to be predicted in terms of the features *Stability, Error, Sign, Wind, Magnitude and Visibility.*

A decision tree is constructed from the set of cases using the decision tree construction algorithm ID3. The tree constructed from the shuttle cases is shown in Figure 1. The decision tree is then used to construct the user querying and classification rule sets. The basic organization for CLIPS-Induce to presented in Figure 2.

Decision Tree Construction

A decision tree predicts the value of one feature in terms of the values of other features. The process by which a prediction is made using a decision trees is described below.

Using the decision tree in Figure 1, the value of the feature *Landing* will be predicted for the example case shown in Table I. Starting at the top (root) of the decision tree, the value of the feature *Visibility* is checked. Because the value is *Yes* the node at the end of the branch labeled *Yes* is next tested. Since the value for the feature *Stability* is *stab*, the *Error* node is next checked. Traversal of the tree continues down the *not(ss)* branch (because the value of Error is not *ss*), across the *mm* branch and finally down the *nn* branch to the leaf labeled *Manual.* The value for the *Landing* feature, predicted by the decision tree for this case is *Manual.*

ID3 is a decision tree construction algorithm that builds a decision tree consistent with a set of cases. A high-level description of the algorithm is shown in Table II. The tree

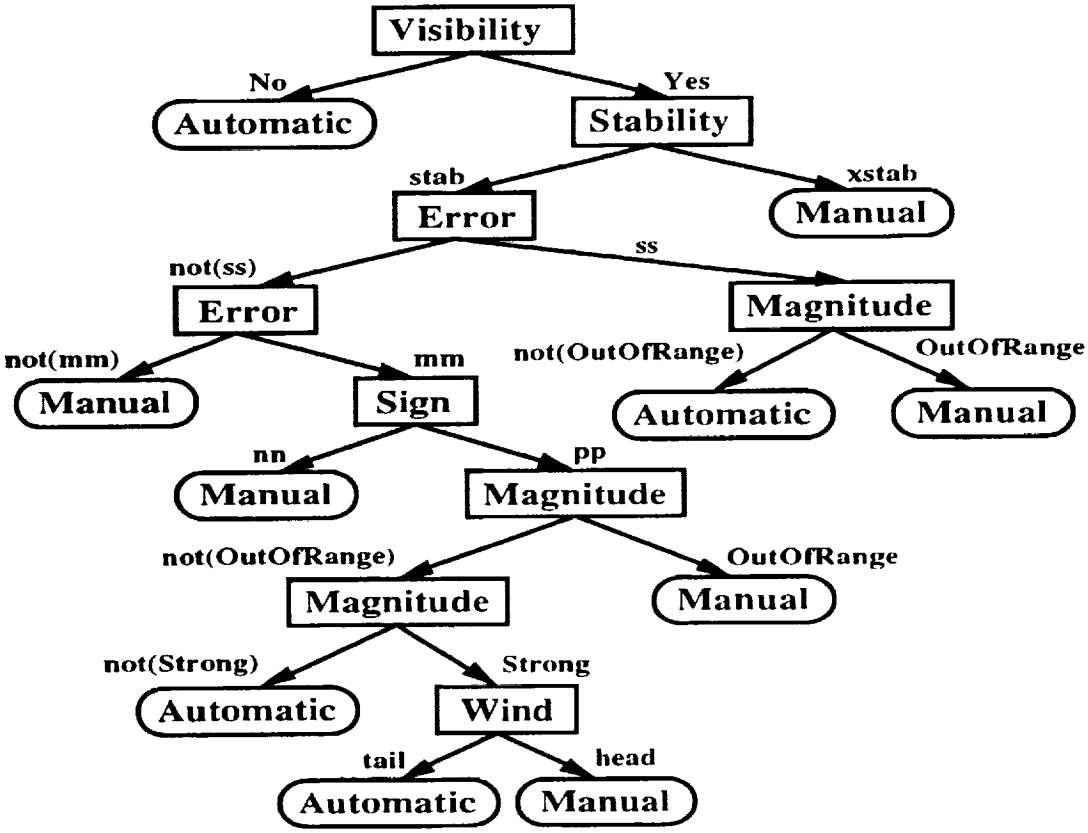


Figure 1: Decision tree constructed by ID3 using the shuttle cases.

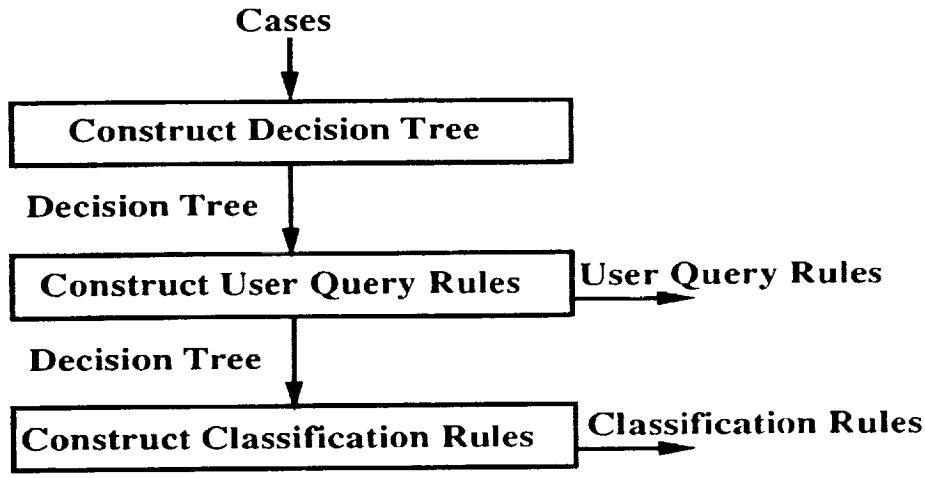


Figure 2: CLIPS-Induce Architecture

is constructed in a recursive top-down manner. At each step in the tree's construction, the algorithm works on the set of cases associated with a node in the partial tree. If the cases at the node all have the same value for the feature to be predicted, the node is made into a leaf. Otherwise, a set of tests is evaluated to determine which test best partitions the set of cases down each branch. The metric used to evaluate the partition made by a particular test is know as information gain (for a more complete description of information gain and ID3, see [1]). Once a test is selected for a node, the cases are partitioned down each branch, and the algorithm is recursively called on the cases at the end of each branch.

Table II: ID3 Decision Tree Construction Algorithm.

```
function generate_dtree(cases)
   if stop_splitting(cases)
      return leaf_class(cases);
   else
      best_cost := eval_examples_cost(cases);
      for all tests()
         cost := eval_cost(cases,test);
         if cost < best_cost then
            best_cost := cost;
            best_test := test;
      for all case_partitions(cases,best_test)
         branches := branches ∪{generate-dtree(case_partition)};
      return (best_test,branches);
```

There are three types of tests that are used by CLIPS-Induce to construct decision trees:

1. Two branch *feature* = *value* test: One branch for *feature* = *value* and a second branch for *feature* $\neq$ *value*.

2. Multi-branch *feature* = *value* test: A branch for each of the values that *feature* has.

3. Two branch *feature* > *value* test: One branch for *feature* > *value* and a second branch for *feature* $\leq$ *value*.

The first and second test types are used for nominal-valued features, e.g. color. Whereas the third test type is used for features with real or ordered values, e.g. age or size.

Rule Generation

The first step in rule generation is to generate the user query rules. The purpose of each user query rule is to ask the user for the value of a particular feature. The user-defined function used to ask the actual questions is shown below.

```
(deffunction ask-question (?question)
  (format t "%s " ?question)
  (read))
```

The query rules are generated such that they only fire when the value for a particular feature is needed and not already available. If, for example, the values for certain features were asserted before execution of the classification rules began, query rules for those features would never fire. Typically, user query rules and classification rules fire in an interleaved manner.

User query rules are generated via a pre-order traversal of the tree. During the traversal, each internal node of the tree is associated with a unique identifier that is used to identify the act of having visited that node during rule execution. An example user query rule, for the *Sign* node in Figure 1, is shown below.

```
(defrule sign-query-g773
  (node node8)
  (not (feature sign ?value))
  =>
  (bind ?answer (ask-question "What is the value of feature sign?"))
  (assert (feature sign ?answer)))
```

The second step in rule generation is to generate the classification rules. The purpose of the classification rules is to traverse the decision tree along a path from the root of the tree to a leaf. Upon reaching the leaf, the value for the feature to be predicted is asserted to the fact-list. Whereas query rules are associated with internal nodes in the decision tree, classification rules are associated with branches in the tree. The two rules for the branches from the *Sign* node in Figure 1, are shown in Table III.

The *Sign* node is identified as *node8*. If the value for feature *Sign* is *pp*, than *(node node9)* will be asserted by the first rule. *Node9* is associated with the *Magnitude* node. If the value for *Sign* were instead *nn*, the value *manual* for the predicted feature *Landing* would be asserted by the second rule. In the later case, because no new *(node ...)* fact is asserted, execution of the user query and classification rules halts.

Table III: Example classification rules.

```
(defrule node8-sign-pp
  ?n <- (node node8)
  (feature sign pp)
  =>
  (retract ?n)
  (assert (node node9)))

(defrule node8-sign-nn
  ?n <- (node node8)
  (feature sign nn)
  =>
  (retract ?n)
  (assert (feature landing manual)))
```

ANALYSIS

The first issue to be concerned with in using the CLIPS-Induce, is the time savings relative to generating the rules by hand. For the shuttle problem, the 25 rules were generated from a set of 277 cases in only a few seconds. For another problem that deals with predicting lymph node cancer, 87 rules were generated in less than a minute. Other problems have been observed to generate rule-bases with as many as 500 rules in very reasonable amounts of time. Given that a set of test cases is available, CLIPS-Induce can save a great deal of time.

The second issue concerns the accuracy of the induced rules on new cases. This concern has been addressed by the area of machine learning where a great deal of research has been done on the induction of decision trees from cases. Specifically, ID3 has been empirically shown to do well at generating decision trees that are accurate on unseen cases. For example, Figure 3 shows a learning curve for the shuttle problem. Learning curves show the accuracy of a model (a decision tree) on unseen cases, as a function of the number of cases used to generate the model. For the shuttle problem, when only 10% of the 277 cases were used to generate the decision trees, the accuracy on the remaining 90% of the cases is approximately 92%. As the proportion of cases increases, the accuracy of the constructed decision trees increases.

The third and final issue concerns the availability of a sufficient numbers of cases needed to induce an accurate set of rules (from Figure 3, the fewer the number of cases, the less accurate is the induced decision tree). In answer to this concern, even if there are only a small number of cases for a problem, the rule-base generated by CLIPS-Induce can be used as a starting point for a domain expert.

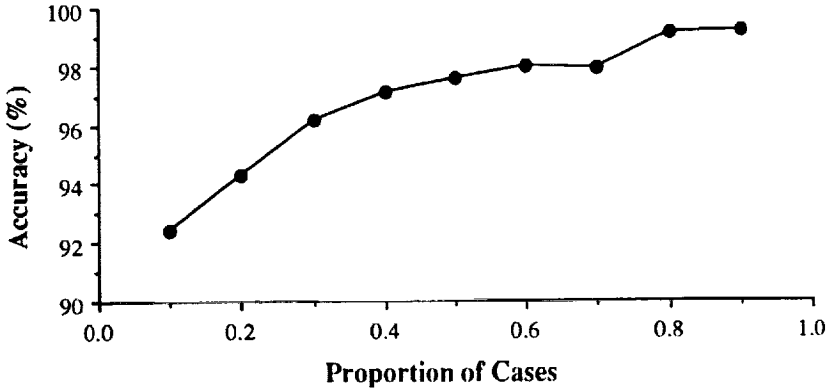


Figure 3: Average accuracy of decision trees as a function of the proportion of the 277 cases used to construct the decision trees. The accuracy of each decision tree is based on the cases not used to construct the tree.

EXTENSIONS

One of the ways that CLIPS-Induce could be extended would be to take advantage of ID3's approach for dealing with missing feature values. Currently, the rule-bases, generated by CLIPS-Induce, halt when the user cannot enter a value for a required feature. The only drawback to extending CLIPS-Induce in this manner, is the increased complexity and reduced understandability of the generated rules.

Another enhancement to CLIPS-Induce would be to use a more sophisticated *ask-question* function. User-query rules could be generated that also pass the set of allowable values or value type to the *ask-question* function. The extra argument could provide constraints on the allowable responses made by the user.

The third extension to CLIPS-Induce would be to allow interactive creation of decision trees. It is often the case that an expert in the field has knowledge that could help in forming a more accurate and more understandable decision tree.

CONCLUSION

In this paper, CLIPS-Induce, a Common Lisp application that induces a CLIPS classification rule-base from a set of test cases, is described. Given a set of test cases, described in terms of a fixed set of features, a decision tree is constructed using the decision tree construction algorithm, ID3. From the decision tree, two sets of rules are extracted. One set of rules, the user query rules, ask the user for the values of features needed to make a classification. The other set of rules, the classification rules, simulate a traversal of the decision tree in order to make the prediction that the decision tree would make. The rule-base formed by CLIPS-Induce can easily be embedded in rule-bases that need classification rule groups.

REFERENCES

1. Quinlan, J.R., "Induction of Decision Trees," MACHINE LEARNING, Boston, Massachusetts, 1(1), 1986, 81-106.

AUTOMATED REVISION OF CLIPS RULE-BASES

Patrick M. Murphy, pmurphy@ics.uci.edu
Michael J. Pazzani, pazzani@ics.uci.edu

Department of Information & Computer Science
University of California, Irvine, CA 92717

Abstract

This paper describes CLIPS-R, a theory revision system for the revision of CLIPS rule-bases. CLIPS-R may be used for a variety of knowledge-base revision tasks, such as refining a prototype system, adapting an existing system to slightly different operating conditions, or improving an operational system that makes occasional errors. We present a description of how CLIPS-R revises rule-bases, and an evaluation of the system on three rule-bases.

INTRODUCTION

Considerable progress has been made in the last few years in the subfield of machine learning known as theory revision, e.g. [1,2,3]. The general goal of this area is to create learning models that can automatically update the knowledge base of a system to be more accurate on a set of test cases. Unfortunately, this progress has not yet been put into common practice. An important reason for the absence of technology transition is that only a restricted form of knowledge bases have been addressed. In particular, only the revision of logical knowledge bases that perform classification tasks with backward chaining rules [4] has been explored. However, nearly all deployed knowledge-based systems make use of forward-chaining production rules with side effects. For example, two of the knowledge-based systems reported on at the 1993 Innovative Applications of Artificial Intelligence use CLIPS. The remainder of the knowledge-based systems use ART, a commercial expert system that has many of the same features as CLIPS.

There are a variety of practical reasons why the production rule formalism is preferred to the logical rule formalism in deployed expert systems. First, production rules are suitable for a variety of reasoning tasks, such as planning, design and scheduling in addition to classification tasks that are addressed by logical rules. Second, most deployed knowledge-based systems must perform a variety of computational activities such as interacting with external databases or printing reports in addition to the "reasoning" tasks. The production system formalism allows such procedural tasks to be easily combined with the reasoning tasks. Third, the production rule systems tend to be computationally more efficient. The production systems allow the knowledge engineer

to have more influence over the flow of control in the systems allowing the performance to be fine tuned. Whereas in a logical system, the rules indicate what inferences are valid, in a production system, the rules indicate both which inferences are valid and which inferences should be made at a particular point.

The revision of CLIPS rule-bases presents a number of challenging problems that have not been addressed in previous research on theory revision. In particular, rules can retract facts from working memory, display information, and request user input. New opportunities to take advantage of additional sources of information also accompany these new problems. For example, a user might provide information that a certain item that was displayed should not have been, or that information is displayed in the wrong order.

In the remainder of this paper, we give an overview description of CLIPS-R, a system for revising CLIPS rule-bases and an evaluation of CLIPS-R on three rule-bases.

DESCRIPTION

CLIPS-R has an iterative refinement control structure (see Figure 1). The system takes as input a rule-base and a set of instances that define constraints on the correct execution of the rules in the rule-base. While there are unsatisfied constraints CLIPS-R heuristically identifies a subset of similar instances as problem instances (instances with many unsatisfied constraints). Using the problem instances, a set of potential repairs to the rule-base are heuristically identified. Each of these repairs is used to temporarily modify the rule-base, with each modified rule-base evaluated over all instances. The repair that improves the rule-base most is used to permanently modify the rule-base. If no repair can increase the evaluation of the rule-base, than the addition of a new rule through rule induction is attempted. If rule induction cannot generate a rule that can improve the evaluation of the rule-base, the revision process halts and the latest rule-base is returned. The process of modification and rule induction continues until all constraints are satisfied or until no progress is made.

Instances

Each instance has two components: initial state information and constraints on the execution of the rule-base given the initial state. The initial state information consists of a set of initial facts to be loaded into the fact-list before execution of the rule-base, and a set of bindings that relate input function calls to their return values. From the automobile diagnosis rule-base, the function *ask-question* with argument "What is the surface state of the points?" may return "burned" for one instance and "contaminated" for another instance. The set of constraints on the execution of the rule-base includes constraints on the contents of the final fact-list (the final fact-list is the fact-list when execution of the rule-base halts), and constraints on the ordering of observable actions such as displaying data or asking the user questions.

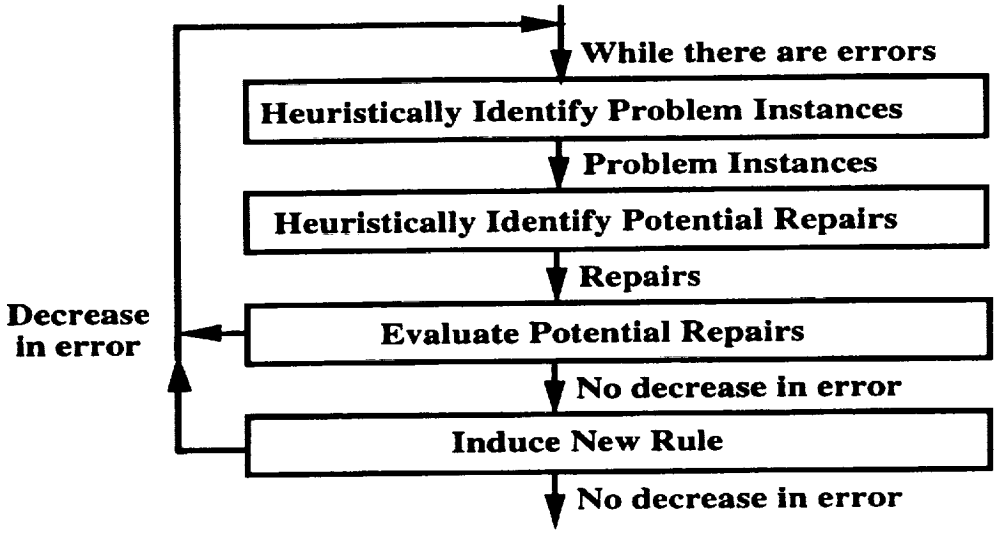


Figure 1: System Organization.

Rule-Base Evaluation

The metric used to evaluate of the rule-base (relative to a set of instances) is the mean error rate across all instances. The error rate for an instance is the percentage of constraints unsatisfied by the execution of the rule-base. An instance is executed in the following manner.

- Reset the rule-base.
- Assert any initial facts.
- Associate bindings with user-defined functions.
- Execute the rule-base until either the agenda is empty or until a user-defined rule execution limit is reached.

During execution of the rule-base for a particular instance, trace information is recorded that is used to determine how many of the constraints associated with the instance are unsatisfied.

Repair Operators

The set of potential repairs to a rule-base are, LHS specialization and generalization (the addition and deletion of conditional elements to the LHS of a rule), action promotion and demotion (the decrease and increase of the embeddedness of an action within if-than-else function, salience modification, assert and retract addition and deletion, observable action modification, rule deletion and rule induction.

Repair Identification

Below is a brief example of how repairs for a particular problem instance are identified. For a more thorough description of these and other aspects of CLIPS-R, see [5]. Assume a problem instance is identified that has an error because the final fact-list contains the extra fact *(repair "add gas")*. The heuristics that suggested repairs for an extra fact are described below.

- The action within the rule that asserted this fact should be deleted.
- Add a retract for the fact to a previously fired rule.
- For each unfired rule in the rule-base that has a retract that could retract this fact, identify repairs that would allow that rule to fire.
- Identify repairs that could cause the rule that asserted the fact to not fire.

EMPIRICAL ANALYSIS

A series of experiments were designed to analyze various characteristics of CLIPS-R. With the first rule-base, automobile diagnosis (distributed with CLIPS), we perform experiments to determine how well CLIPS-R does at increasing the accuracy of randomly mutated rule-bases. For the second domain, we deal with the nematode identification rule-base. With this rule-base, we show how CLIPS-R can be used to extend a rule-base to handle new cases. For the final rule-base, student loan [2], a problem translated from PROLOG to CLIPS, we chow that CLIPS-R is competitive with an existing revision system, FOCL-FRONTIER [6], that is designed to revise the Horn clause rule-bases.

Automobile Diagnosis Rule-Base

The auto diagnosis rule-base is a rule-base of 15 rules. It is an expert system that prints out an introductory message, asks a series of questions of the user, and prints out a concluding message including the predicted diagnosis.

Cases were generated from 258 combinations of responses to the user query function. Each instance consisted of a set of answers for each invocation of the query function as initial state information, a single constraint on the final fact-list and an ordering constraint for the sequence of printout actions. The target constraint for each instance was a positive constraint for a repair fact, e.g. *(repair "Replace the points")*.

Execution of an instance for the auto diagnosis rule-base consisted of clearing the fact-list, setting the bindings that determine the return values for each function call instance (to simulate user input for the user query function) and executing the rule-base to completion. The bindings that associate function calls to their return values allowed an otherwise interactive rule-base to be run in batch mode. This is necessary because

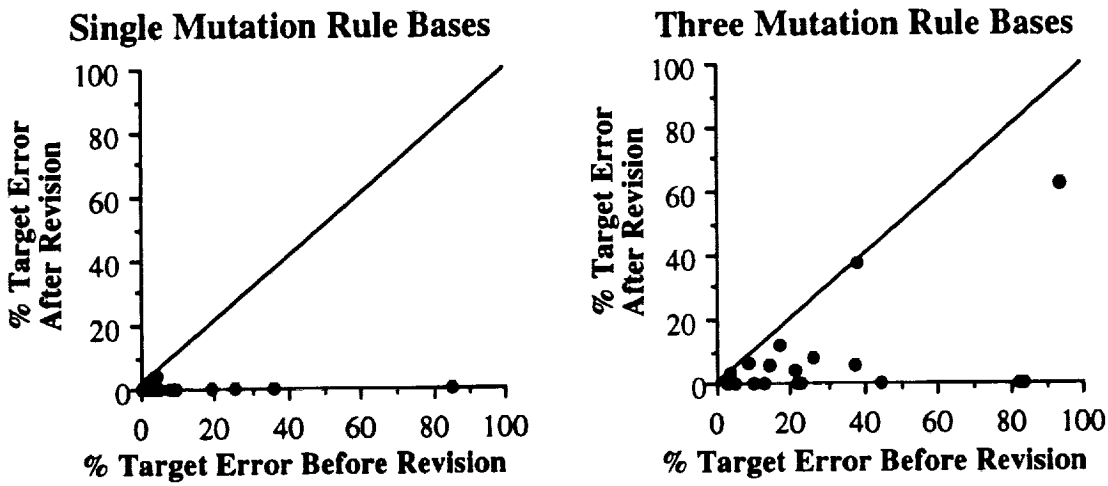


Figure 2: Target Error After Revision as a function of Target Error Before Revision.

no user would be willing to answer the same questions for 50 instances on different variations of the rule-base.

The experiments performed using the auto diagnosis rule-base were designed to determine how well CLIPS-R could do at revising mutated versions of the correct rule-base. Mutations consisted of extra, missing or incorrect conditional elements or actions and incorrect rule salience values. Two sets of 20 mutated rule-bases were randomly generated with one set of rule-bases having only a single mutation and the other set having three mutations per rule-base. Each mutated rule-base was revised using a random set of 50 training instances. The remaining instances were used for testing. Figure 2 contains a scatter plot showing the initial error of each mutated rule-base and the final error after revision of the rule-base.

An analysis of the scatter plots in Figure 2 shows that, for the most part, CLIPS-R is able to reduce the error rates of the mutated rule-bases (points below the diagonal indicate a decrease in error). For one mutation, the average rule-base error was 11.2% before learning and 0.7% after learning. With three mutations, the error before learning was 28.0% and after learning it was 7.2%.

Nematode Identification Rule-Base

The second rule-base, nematode identification (a nematode is a class or phylum of worm), has 93 rules. The intent in presenting this rule-base, is show an example of how CLIPS-R can be used to extend a classification rule-base to handle new cases. The basic requirements for this problem are a rule-base, a set of cases that the rule-base correctly classifies, and a set of cases that are not correctly classified by the rule-base.

For the nematode rule-base, because no cases were provided with the original rule-base, a set of 50 cases were generated by interactively running the rule-base over different

responses to the user query function. In order to simulate a rule-base that is in need of extension, two errors were introduced into the rule-base. Specifically, one rule was deleted (a rule that would normally assert the classification for two of the 50 cases), and a second rule was changed, so that it fired and asserted an incorrect classification for the two cases no longer classified by the deleted rule, see Table I. The two cases that are misclassified by the mutated rule-base, are the cases that CLIPS-R needs to extend the rule-base to cover.

Table I: Mutated Rules.

```
(defrule Pratylenchus
  ?f1 <- (esophagus-glands-overlap-intestine ventrally)
  ?f2 <- (ovary 1)
  =>
  (retract ?f1)
  (retract ?f2)
  (assert (nematode pratylenchus))
  (assert (id-criteria "1. esophagus glands overlap intestine ventrally."
                       "2. ovary 1."
                       "3. head-shape low and flat.")))
```

(a) Deleted rule.

```
(defrule Hirshmanniella
  ?f1 <- (esophagus-glands-overlap-intestine ventrally)
  ;;; ?f2 <- (ovary 2)
  =>
  (retract ?f1)
  ;;; (retract ?f2)
  (assert (nematode hirshmanniella))
  (assert (id-criteria "1. esophagus glands overlap intestine ventrally."
                       "2. ovary 2."
                       "3. head-shape low and flat.")))
```

(b) Rule with deleted conditional element *(ovary 2)* and retract.

When provided with the set of 50 cases and the mutated rule-base, CLIPS-R extends the rule-base to handle the new cases as follows. First, the two misclassified cases are identified by CLIPS-R as the problem cases. Second a set of repairs are identified and evaluated over all 50 cases. After completing the evaluations the repair that specialized the rule *Hirshmanniella* to include *(ovary 2)* as a conditional element is selected as the best repair because it is the repair that most decreased the error rate over all 50 cases. Upon permanently adding *(ovary 2)* to the rule *Hirshmanniella*, the two cases,

previously misclassified, are unclassified by the revised rule-base. After completion of a second set of repair evaluations with no success at reducing error rate, rule induction is successfully used to fix the unclassified cases, see Table II.

Table II: Revised Rules.

```
(defrule G123091
  (esophagus-glands-overlap-intestine ventrally)
  (ovary 1)
  =>
  (assert (nema-id pratylenchus)))
```

(a) New rule.

```
(defrule Hirshmanniella
  ?f1 <- (esophagus-glands-overlap-intestine ventrally)
  (ovary 2)
  =>
  (retract ?f1)
  (assert (nematode hirshmanniella))
  (assert (id-criteria "1. esophagus glands overlap intestine ventrally."
                       "2. ovary 2."
                       "3. head-shape low and flat.")))
```

(b) Revised rule with conditional element *(ovary 2)* added.

Note the difference between the original rules shown in Table I and the revisions of the mutated rules shown in Table II. The revised *Hirshmanniella* rule differs from the original rule by the absence of a retract for the fact matching the *(ovary 2)* conditional element. The set of 50 test cases were insufficient to recognize that a retract was missing. A similar problem is true for the induced rule *G123091*. This rule was was added by CLIPS-R to take the place of the the deleted rule *Pratylenchus*. While this rule asserts a classification that is correct with respect to the test cases, *(nema-id pratylenchus)*, it is not quite the same assertion made by the deleted rule, *(nematode pratylenchus)* (if *(nematode pratylenchus)* had been asserted by *G123091*, it would later be replaced by the fact *(nema-id pratylenchus)*). In short, the results of this experiment highlight the need for a comprehensive library of test cases.

Student Loan Rule-Base

In the original form, the student loan domain consists of a set of nine rules (represented as Horn clauses) and a set of 1000 cases. The rule-base contains four errors (an extra

literal, a missing literal, an extra clause and a missing clause). The initial theory has an error of 21.6%. In order to use this rule-base with CLIPS-R, the nine Horn clause rules were converted into nine production rules, each with a single assert action. Multiple clauses in the Horn clause rules were converted to a disjunction of conjuncts within a CLIPS production rule.

Execution of a case for the student loan rule-base consisted of asserting into an empty fact-list a set of facts specific to the case and then executing the rule-base to completion. All results for the following experiments are averages of 20 runs. All cases not used for training are used for testing.

Table III: A Comparison of FOCL-FRONTIER and CLIPS-R.

Num. Cases	FOCL % Error	CLIPS-R % Error
25	11.8	12.6
50	5.8	3.0
75	2.8	2.1

The experiment performed was to determine how well CLIPS-R performed at revising the rule-base relative to FOCL-FRONTIER. Table III shows that the error rate is competitive with that of FOCL-FRONTIER on this problem. Only with 50 training examples is the difference in error significant ($p < .05$).

FUTURE WORK

CLIPS-R is still in its infancy and we expect many of the details of the individual operators and heuristics to change as this work matures. Future directions include solutions to the issues that arose when revising the nematode rule, e.g. a better language for representing constraints on the correct execution of the rule-base, and the use of rule clustering, rule-models and rule-groups to guide the revision and induction of rules. Additional research could include a greater understanding of the distributions of rule-base coding styles, automated rule-base understanding systems, and revision strategies that simulate the methods by which humans manually revise rule-bases.

CONCLUSION

We have described CLIPS-R, a theory revision system for the revision of CLIPS rule-bases. Novel aspects of CLIPS-R include the ability to handle forward chaining theories with "nonlogical" operations such as rule saliences and the retraction of information from working memory. The system is organized with an iterative refinement control structure that identifies a set of similar problematic instances, identifies repairs that can fix the errors associated with the instances, and than evaluates each repair to identify

the repair that best improves the rule-base. CLIPS-R can take advantage of a variety of user specified constraints on the correct processing of instances such as ordering constraints on the displaying of information, and the contents of the final fact-list. In addition, CLIPS-R can operate as well as existing systems when the only constraint on processing an instance is the correct classification of the instance.

ACKNOWLEDGMENTS

The research reported here was supported in part by NSF Grant IRI-9310413, ARPA Grant F49620-92-J-0430, and AFOSR AASERT grant F49620-93-1-0569.

REFERENCES

1. Ourston, D. & Mooney, R., "Changing the Rules: A Comprehensive Approach to Theory Refinement," Proceedings of the Eighth National Conference on Artificial Intelligence, AAAI-90, 1990, 815-820.

2. Pazzani, M.J. & Brunk, C., "Detecting and Correcting Errors in Rule-Based Expert Systems: An Integration of Empirical and Explanation-Based Learning," In Knowledge Acquisition, 3, 1991, 157-173.

3. Wogulis, J. & Pazzani, M.J., "A Methodology for Evaluating Theory Revision Systems: Results with AUDREY II," Proceedings of the 13th International Joint Conference on Artificial Intelligence, IJCAI93, 1993, 1128-1134.

4. Clancey, W., "Classification problem solving," Proceedings of the National Conference on Artificial Intelligence, 1984, 49-55.

5. Murphy, P.M. & Pazzani, M.J., "Revision of Production System Rule-Bases," Machine Learning: Proceedings of the Eleventh International Conference, 1994, 199-207.

6. Pazzani, M.J. & Brunk, C., "Finding Accurate Frontiers: A Knowledge-Intensive Approach to Relational Learning," Proceedings of the Eleventh National Conference on Artificial Intelligence, AAAI93, 1993, 328-334.

omit

Session 7A: Distributed Processing and Virtual Reality Extensions

Session Chair: Mark Engelberg

DAI-CLIPS: Distributed, Asynchronous, Interacting CLIPS

Denis Gagné & Alain Garant

Groupe de Recherche en Intelligence Artificielle Distribuée

Collège militaire royal de Saint-Jean

Richelain, (Québec)

Canada, J0J 1R0

dgagne@cmr.ca

Abstract

DAI-CLIPS is a distributed computational environment within which each CLIPS is an active independent computational entity with the ability to communicate freely with other CLIPS. Furthermore, new CLIPS can be created, others can be deleted or modify their expertise, all dynamically in an asynchronous and independent fashion during execution. The participating CLIPS are distributed over a network of heterogeneous processors taking full advantage of the available processing power. We present the general framework encompassing DAI-CLIPS and discuss some of its advantages and potential applications.

1 Introduction

Scenarios to be solved by Artificial Intelligence (AI) applications rapidly increase in complexity. If we are to even allude to the enormously difficult endeavor that represents Artificial Intelligence, very flexible, robust and powerful computational systems are going to be needed. These programs, and the processing architecture supporting them, will have to be able to cope with a very wide range of dynamic external demands that are simply unpredictable.

Conditions vary greatly across tasks instances and constantly dictate different accomplishment strategies and usage of disparate sources of expertise. To be effective in such situations, fusion must take place at many levels (information, expertise, processes,...). Maximum degree of openness and flexibility is required of AI systems. Both hardware architecture and software solutions must allow for scalable performance and scalable functionalities. This, in return, will allow the matching of AI systems to the needs of the situation as well as gaining access to the latest technological advances.

We believe that complex problems can best be solved via a pendemonium of smaller agents. Each agent specializes in a different narrow aspect of cognition or field of knowledge [Min85, Ten88]. Emphasis is thus placed on data and application parallelism.

The computational environment presented herein integrates the simplicity, elegance and expressiveness of the ACTOR computational model [Hew73, Agh86] with the exceptional processing power of heterogeneous distributed and parallel architectures [Des93]. Expressiveness is increased by providing for disjunct or even disparate sources of expertise to cohabitate rather than trying to integrate them. The interfacing or fusion required is achieved via message passing enabling asynchronous exchange of knowledge (facts, rules) among agents. Agents in the present context are effectively extended CLIPS. CLIPS is an expert system tool that was developed by NASA at the Software Technology Branch of the Johnson Space Center [Gia94].

DAI-CLIPS is a distributed computational environment within which each extended CLIPS is an active independent computational entity with the ability to communicate freely with other CLIPS. Furthermore, new CLIPS can be created, others can be deleted or modify their expertise, all dynamically in a totally asynchronous and independent fashion during execution.

The remainder of this text is structured as follows. Section 2 highlights the key characteristic of the Actor computational model that influenced DAI-CLIPS and briefly describes the system layer that constitutes the foundation of DAI-CLIPS. In section 3, we provide a description of the conceptual framework offered by DAI-CLIPS by outlining its global functionalities. In section 4, we provide some details about the architecture and available functions. Section 5 brushes a quick picture of a few potential areas of research and development that could benefit from such computational environment. Finally, sections 6 and 7 provides a discussion and our conclusions on the subject.

2 The Actor Model & CLAP

In this section we highlight some of the main characteristics of the Actor model that influenced and characterizes DAI-CLIPS. We then present a brief description of CLAP [Des93, Gag93, Gag94], a system layer based on the Actor Model, that constitute the foundation of DAI-CLIPS.

2.1 The Actor Model

A detailed description of the Actor Model can be found in [Agh86, Hew77]. We will only discuss here a few of the more salient characteristics of the model:

Distributed. The Actor model consists of numerous independent computational entities (actors). The actors process information concurrently, which permits the overall system to handle the simultaneous arrival of information from different outside sources.

Asynchronous. New information may arrive at any time, requiring actors to operate asynchronously. Also, actors can be physically separated where distance prohibits them from acting synchronously. Each actor has a mailbox (buffer) where messages can be stored while waiting to be processed.

Interactive. The Actor model is characterized by a continuous information exchange through message passing, subject to unanticipated communication from the outside.

Thus, an actor is basically an active independent computational entity communicating freely with other actors.

There are at least two different ways to look at the Actor model. Viewed as a system, it is comprised of two parts: the actors and a system layer (operating/management system). From such a point of view the actors are the only available computational entities. The system layer is responsible to manage, create and destroy actors as required or requested. The system layer is also responsible for ensuring message passing among actors.

Viewed as a computational entity, an actor also comprises two parts: a script, that defines the behaviors of the actor upon receipt of a message; and a finite set of acquaintances which are the other actors known to the actor.

2.2 CLAP

The above viewpoint duality is preserved in CLAP[1]. CLAP is an implementation of an extension of the actor model that can execute on a distributed heterogeneous network of processors. The present version of CLAP can execute over a network of SUN SPARC workstations and Alex Informatique AVX parallel machines which are transputer based distributed memory machines [Des93]. A port to HP and SGI workstations is in progress.

CLAP is an object-oriented programming environment that implements the following concepts of the Actor model: the notion of actor, behaviors, mailbox, and parallelism at the actor level. Further, CLAP offers the extension to the model of intra-actor parallelism.

Generally, CLAP applications will consist of many programs distributed over available processors executing as a task under the control of the CLAP run time environment. In CLAP, each actor is a member of a given task. It is up to the programmer to determine how many actors there will be for any given task (although, a large number of actors in a single task could mean the loss of potential parallelism in the application.) A scheduler controls the execution of processes inside the tasks. Each task possesses a message server that handles message reception for the actors in the task. Inter-processor message transmissions are handled via RPC servers. XDR filters and type information are utilized for the encoding and decoding of these messages. The CLAP environment is implemented in C++.

3 The Conceptual Framework

DAI-CLIPS is a distributed computational environment within which each CLIPS has been extended to become an active independent computational entity with the ability to communicate freely with other extended CLIPS. Furthermore, new extended CLIPS can be created, others can be deleted or modify their expertise, all dynamically in a totally asynchronous and independent fashion during execution.

The *desirata* behind DAI-CLIPS is to produce a flexible development tool that captures the essence of the "aggregate of micro agents" thesis supported by many in the study of Computational Intelligence and Cybernetic [Hew73, Min85, Ten88]. The underlying thesis being to have "micro agents", in our case complete CLIPS, specialized in different very

[1] C++ Library for Actor Programming.

narrow aspects of cognition or fields of knowledge[2]. As they go about their tasks, these micro-agents confer with each other and form coalitions producing collated, revised enhanced views of the raw data they take in. These coalitions and their mechanism implement various cognitive processes leading to the successful resolution of the problem.

3.1 D.A.I.

The three highlighted characteristics of the Actor model in the previous section, namely distributed, asynchronous and interactive, are at the basis of the conceptual framework for DAI-CLIPS. The augmented CLIPS participating in the environment are completely encapsulated and independent allowing their distribution at both the software and hardware level. Meaning that not only can the CLIPS execute in parallel but they can also be physically distributed over the network of available processors. Our present version of DAI-CLIPS can have participating CLIPS distributed over a network of SPARC workstations and/or the nodes of a transputer based distributed memory parallel machine. The interaction among the CLIPS is asynchronous (synchronicity can be imposed when required). The interchange of knowledge between these extended CLIPS can involve exchanging facts, rules, and any other CLIPS data object or functionality.

3.2 Cooperation

The DAI-CLIPS environment is conducive of cooperation among a set of independent CLIPS. We regard as cooperation any exchange of knowledge among CLIPS whether productive or not.

There is *a priori* no pre-defined notion of an organizational structure among the CLIPS in DAI-CLIPS. Any desired type of organization (*e.g.* hierarchy, free market, assembly line, task force, *etc.*) can be achieved by providing each CLIPS the appropriate knowledge of the structure and the mechanism or protocol to achieve it.

The broad definition of cooperation and the inexistence of pre-defined organizational structures in DAI-CLIPS were conscious initial choices. We wanted to maintain the highest flexibility possible for the environment in this first incarnation. We are contemplating the introduction of mechanisms to DAI-CLIPS to ease the elaboration of specific types of organizations based on the premise of groups or aggregates. The aim of these efforts is to capture the recursive notion of *agency*.

3.3 Dynamic Creation

A powerful capability of DAI-CLIPS is the possibility of dynamically generating or destroying participating CLIPS at run time. The generation of new CLIPS can involve introducing a new expertise or simply cloning an existing participant. When generating a new CLIPS, one can specify which expertise the CLIPS is to possess by indicating the appropriate knowledge base(s) to be loaded in the CLIPS at creation. This functionality has enormous potentials that we have yet to completely explore.

[2]Expert Systems excel under these domain constraints.

4 The Architecture

The general framework encompassing DAI-CLIPS can be viewed as four layers: the hardware layer, the system layer, the agent layer, and the application layer (see fig1). The hardware layer consists of a set of nodes (available processors) on the network (SPARCs and transputers). The system layer (CLAP) is responsible to manage, create and destroy the processes required or requested from the above agent layer as well as managing inter-agent communications. The agent layer provides a series of functionalities to implement various cognitive processes via coalitions and organization mechanisms for a series of specialized micro-agents (DAI-CLIPS). Finally, the application layer provides interfacing services and captures and implements the user's conceptualization of the targeted domain. Such conceptualization usually involves the universe of discourse or set of objects presumed in the domain, a set of functions on the universe of discourse, and a set of relations on the universe of discourse [Gen87].

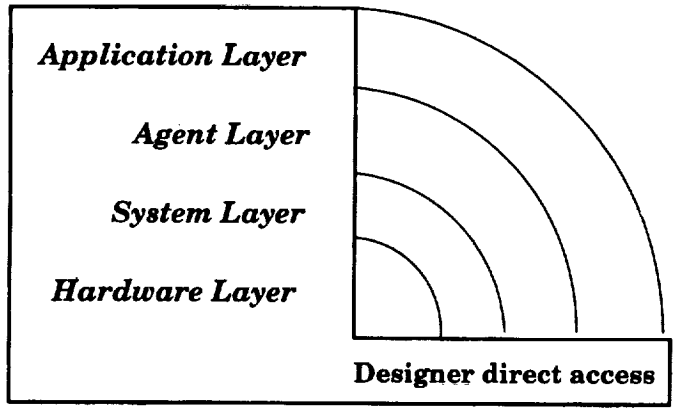


Figure 1: The conceptual framework.

Within this general framework, an application designer can directly access and manipulate any of the four layers of the environment. This provides the designer with the flexibility of manipulating objects at the level of abstraction he is more at ease with. For example, a more advanced application designer could seek efficiency in his particular application by manipulating objects all the way down to the system layer level, where someone else may be quite content of the functionalities provided at the top layer.

4.1 Design

In the present version of the environment, each augmented CLIPS is associated with a CLAP actor. These actors are loaded on different available processing nodes according to the load of the nodes. To each augmented CLIPS is connected an interface which provides access to the individual standard command loops of the CLIPS. An initial knowledge base is loaded in each CLIPS (see fig2). Note that it is possible for two CLIPS to be uploaded with the

same initial knowledge base or for a CLIPS to upload a supplementary knowledge base at run time.

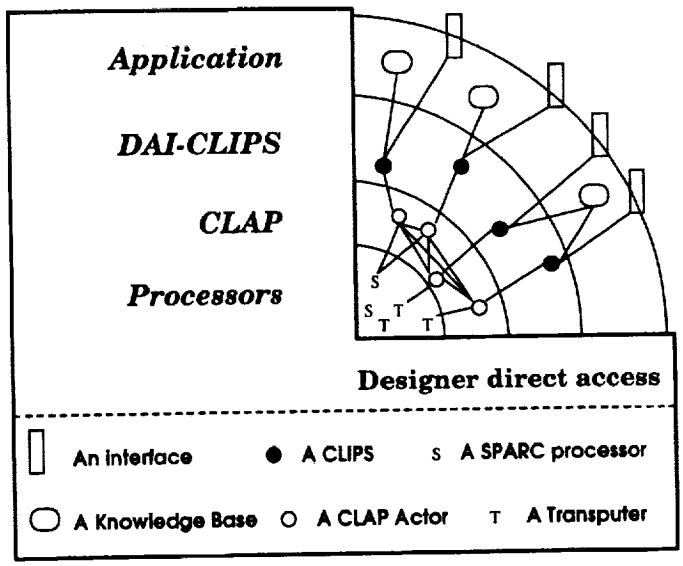


Figure 2: Surrounding environment of DAI-CLIPS.

4.2 Implementation

In this section we enumerate some of the functions specific to DAI-CLIPS and describe their respective use and functionality. The list is not exhaustive[3], rather the intent here is to present some of the main functions which can be used directly by the user.

create-agent

Purpose: Creates a named agent without expertise.
Synopsis: `(create-agent <string-or-symbol-agent-name>)`
Behavior: The agent `<string-or-symbol-agent-name>` is created.

destroy-agent

Purpose: Destroys a named agent.
Synopsis: `(destroy-agent <string-or-symbol-agent-name>)`
Behavior: The agent `<string-or-symbol-agent-name>` is destroyed.

send-fact-agent

Purpose: Asserts a run-time fact in a named agent.
Synopsis: `(send-fact-agent <string-or-symbol-agent-name> <string>)`
Behavior: The fact `<string>` is asserted in the agent `<string-or-symbol-agent-name>` who then executes.

[3]Due to restricted space in this article.

send-deffact-agent

Purpose: Defines a persistent fact in a named agent.

Synopsis: `(send-deffact-agent <string-or-symbol-agent-name> <symbol-deffact-name>)`

Behavior: The fact `<symbol-deffact-name>` is permanently asserted in the agent `<string-or-symbol-agent-name>` who then executes.

send-defglobal-agent

Purpose: Defines a global variable in a named agent.

Synopsis: `(send-defglobal-agent <string-or-symbol-agent-name> <symbol-defglobal-name>)`

Behavior: The global variable `<symbol-defglobal-name>` is defined in the agent `<string-or-symbol-agent-name>` who then executes.

send-defrule-agent

Purpose: Defines a rule in a named agent.

Synopsis: `(send-defrule-agent <string-or-symbol-agent-name> <symbol-defrule-name>)`

Behavior: The rule `<symbol-defrule-name>` is added in the expertise of agent `<string-or-symbol-agent-name>` who then executes.

send-deftemplate-agent

Purpose: Defines a template in a named agent.

Synopsis: `(send-deftemplate-agent <string-or-symbol-agent-name> <symbol-deftemplate-name>)`

Behavior: The template `<symbol-deftemplate-name>` is added in the expertise of agent `<string-or-symbol-agent-name>` who then executes.

load-for-agent

Purpose: Send a message to a named agent ordering him to load a specific expertise from a named file.

Synopsis: `(load-for-agent <string-or-symbol-agent-name> <file-name>)`

Behavior: The agent `<string-or-symbol-agent-name>` possesses the expertise specified in `<file-name>`.

5 Potential Areas of Applications

DAI-CLIPS provides an environment with a varying number of autonomous knowledge based systems (expert-systems) that can exchange knowledge asynchronously. Such organizations of interconnected and independent computational systems are what Hewit calls **Open Systems**[4] [Hew85]. We thus refer to DAI-CLIPS as an *Open Knowledge Based Environment* or *Open KBE* for short. The potential areas of research and development that could benefit from such a computational environment are considerable.

[4]The term "open system" being an overloaded term, we specifically refer to Hewit's definition of open systems within the context of this article.

5.1 Distributed Artificial Intelligence

The first such area that comes to mind is that of Distributed Artificial Intelligence (DAI) [Bon88, Gas89, Huh87]. The facilities available in DAI-CLIPS to support interaction between "intelligent" agents make it a flexible tool for DAI applications and research. In comparison with some existing DAI test beds, DAI-CLIPS: does not impose a specific control architecture such as the blackboard in GBB [Cor86]; does not restrict agents to a specific set of operators as in TRUCKWORLD; and is not a domain specific simulator as in PHOENIX[Han93]. The most closely related work is SOCIAL CLIPS [Adl91]. The major difference with SOCIAL CLIPS is DAI-CLIPS' dynamic creation and destruction of participating CLIPS at run time.

There are *a priori* no predefined domain of application for DAI-CLIPS. A designer is free to specialize his agents in the domain of his/her choice. Further, the agents can be heterogeneous in their speciality (expertise) within a single application. The only imposed commonality in DAI-CLIPS is the use of the extended CLIPS shell. The added power provided by the dynamic creation/destruction of agents within DAI-CLIPS is the source of the potential area of application proposed in the next section.

5.2 Evolutionary Computing

The principle behind evolutionary computing is that of population control [Koz93]. That is ensuring that the population is not allowed to grow indefinitely by selectively curtailing it. This population control is carried out by creative and destructive processes guided by natural selection principles. The destructive process examines the current generation (population) and curtails it by destroying its weakest members (usually those with the lowest values from some predefined fitness measure). The creative process introduces a new generation created from the survivors of the destructive process. The expected result is that of a better fit or optimum population.

Given DAI-CLIPS capability of dynamically creating and destroying participating CLIPS at run-time, one can begin to explore the potential of coarse grain evolutionary computing. That is, applying evolutionary computing principles to a population of "agents" or expert systems in order to optain a population of expert systems that selectively better perform on a global task in accordance with some selected fitness measure. The creative and destructive processes could be carried out by two independent agents. One agent evaluating the agents of the population and destroying those that do not perform as expected (destructive process), another, either bringing together the expertise of two fit agents into a newly created expert or simply cloning a fit agent (creative process). We will refer to this approach as a *disembodied genetic mechanism.* Alternatively, the agents of a population could themselves possess "genetic knowledge" that would lead to self-evaluation. Based on the knowledge of its own fitness, an agent could then decide to terminate operations or to seek an appropriate agent for procreation (via mutation, crossover, *etc.*). This *embodied genetic mechanism* could take place based on some pre-determined evolution cycle. Note that both the embodied and disembodied genetic mechanisms can take place continuously and in totally asynchronous fashion.

6 Discussion

Open KBEs such as the one presented herein have considerable advantages:

- they allow independent systems to cooperate in solving problems;
- they allow disparate participant systems to share expertise;
- they allow for the presence of incoherent information among participant systems, no need for global consistency;
- they provide for participant systems to work in parallel on common problems;
- participant systems can be distributed physically to make ultimate use of the available processing power;
- asynchronous communication ensures very remote chances of deadlock;
- fault tolerance is easy to implement via system redundancy;
- participant systems can be developed and implemented independently and modularly;
- participant system are reusable in other applications;

and many others.

By choice, DAI-CLIPS has one limitation with respect to Open KBE: the participant systems are limited to CLIPS based systems. In fact, the general framework encompassing DAI-CLIPS can easily be extended to allow heterogeneous applications (*e.g.* other ES shells, Data Bases, Procedural applications) to participate through the use of a common formal language for the interchange of knowledge among disparate computer programs such as *Knowledge Interface Format* (KIF) and the use of a common message format and message-handling protocol such as the *Knowledge Query and Manipulation Language* (KQML). The use and adherance to these two upcoming standards from the DARPA Knowledge Sharing Initiative can assure that any incompatibility in the paticipant systems' underlying models for representing data, knowledge and commands can be ironed out to attain the desired higher level of openness.

DAI-CLIPS and its encompassing environment will be the source of more research and enhancements. We are presently putting the final touch to a second version of DAI-CLIPS that implements the notion of multiple behaviors from the Actor model. That is, the capability of an agent to change its behavior in order to process the next message. Effectively, a CLIPS shell will possess different expertise and will "context switch" to make use of the appropriate knowledge to process the received message.

7 Conclusion

We introduced DAI-CLIPS, a distributed computational environment within which each CLIPS is an active independent computational entity communicating freely with other CLIPS. Open KBEs such as this one have many advantages, in particular, they allow for

scalable performance and scalable functionalities at both the hardware and the software level. The potential applications of such environments are considerable. The unique power of dynamic creation and destruction of DAI-CLIPS could lead to new forms of "intelligent" evolutionary systems.

8 Acknowledgments

The authors would like to thank Jocelyn Desbiens and the members of the Centre de Recherche en Informatique Distribuée (CRID) for their constant support in the implementation of DAI-CLIPS. We also want to thank Alain Dubreuil and André Trudel for comments on an earlier version of this article.

References

[Adl91] Adler, R., "Integrating CLIPS Applications into Heterogeneous Distributed Systems.", In *Proceedings of the Second CLIPS Conference*, NASA Conference Publication 10085, 1991.

[Agh86] Agha, G.A., *Actors: A Model of Concurrent Computation in Distributed Systems,* Cambridge, Massachusetts: MIT Press, 1986.

[Bon88] Bond, A. & Gasser, L. (Eds), *Readings in Distributed Artificial Intelligence.*, Los Altos, California: Morgan Kaufmann, 1988.

[Cor86] Corkill, D., Gallagher, K. & Murray, K.,"GBB: A Generic Blackboard Development System." In *Proceedings of AAAI-86*, 1986.

[Des93] Desbiens, J., Toulouse, M. & Gagné, D.,"CLAP: Une implantation du modèle Acteur sur réseau hétérogène"., In *Proceedings of the 1993 DND Workshop on Knowledge Based Systems/Robotics.* Ottawa, Ontario, 1993. (In French)

[Gag93] Gagné, D., Nault, G., Garant, A. & Desbiens, J.,"Aurora: A Multi-Agent Prototype Modelling Crew Interpersonal Communication Network"., In *Proceedings of the 1993 DND Workshop on Knowledge Based Systems/Robotics.* Ottawa, Ontario, 1993.

[Gag94] Gagné, D., Desbiens, J. & Nault, G.,"A Multi-Agent System Simulating Crew Interaction in a Military Aircraft"., In *Proceedings of the Second World Congress on Expert Systems.* Estoril, Portugal, 1994.

[Gas89] Gasser, L. & Huhns, M.N. (Eds), *Distributed Artificial Intelligence: Volume II*, Los Altos, California: Morgan Kaufmann, 1989.

[Gen87] Genesereth, M. & Nilsson, N.,*Logical Foundations of Artificial Intelligence.*, Morgan Kaufmann, 1987.

[Gia94] Giarratano, J. & Riley, G.,*Expert Systems: Principles and Programming.*, PWS Publishing Company, 1994.

[Han93] Hanks,S., Pollack, M. & Cohen, P.,"Benchmarks, Test Beds, Controlled Experimentation and the Design of Agent Architectures.", In *AI Magazine*, Vol. 14, No. 4, Winter 1993.

[Hew73] Hewit, C., Bishop, P. & Steiger, R., "A Universal Modular Actor Formalism for Artificial Intelligence.", In *Proceedings of the 3rd Joint Conference on Artificial Intelligence (IJCAI73).*, Stanford, California, 1973.

[Hew77] Hewit, C., "Viewing Control Structures as Pattern of Passing Messages.", In *Journal of Artificial Intelligence.*, Vol 8, No. 3, 1977.

[Hew85] Hewit, C., "The Challenge of Open Systems.", *BYTE Magazine*, Vol. 10, No. 4, April 1985.

[Huh87] Huhns, M.N. (Ed), *Distributed Artificial Intelligence*, Los Altos, California: Morgan Kaufmann, 1987.

[Koz93] Koza, J.,*Genetic Programming.*, The MIT Press, 1993.

[Min85] Minsky, M., *The Society of the Mind.*, Simon and Schuster, New York, 1985.

[Ten88] Tenney, R. & Sandell, JR., "Strategies for Distributed Decisionmaking." In Bond & Gasser (Eds) *Readings in Distributed Artificial Intelligence.*, Los Altos, California: Morgan Kaufmann, 1989.

N95-19764

PCLIPS: PARALLEL CLIPS

Lawrence O. Hall[1], Bonnie H. Bennett[2], and Ivan Tello

Hall & Tello: Department of Computer Science and Engineering
University of South Florida
Tampa, FL 33620
hall@csee.usf.edu

Bennett: Honeywell Technology Center
3660 Technology Driver MN65-2600
Minneapolis, MN 55418
bennett@src.honeywell.com

ABSTRACT

A parallel version of CLIPS 5.1 has been developed to run on Intel Hypercubes. The user interface is the same as that for CLIPS with some added commands to allow for parallel calls. A complete version of CLIPS runs on each node of the hypercube. The system has been instrumented to display the time spent in the match, recognize, and act cycles on each node. Only rule-level parallelism is supported. Parallel commands enable the assertion and retraction of facts to/from remote nodes working memory.

Parallel CLIPS was used to implement a knowledge-based command, control, communications, and intelligence (C^3I) system to demonstrate the fusion of high-level, disparate sources. We discuss the nature of the information fusion problem, our approach, and implementation. Parallel CLIPS has also been used to run several benchmark parallel knowledge bases such as one to set up a cafeteria. Results shown from running Parallel CLIPS with parallel knowledge base partitions indicate that significant speed increases, including superlinear in some cases, are possible.

INTRODUCTION

Parallel CLIPS (PCLIPS) is a rule-level parallelization of the CLIPS 5.1 expert system tool. The concentration on rule-level parallelism allows the developed system to run effectively on current multiple instruction multiple data (MIMD) machines. PCLIPS has been tested on an Intel Hypercube iPSC-2/386 and I860. Our approach bears similarities in focus to research discussed in [12, 6, 7].

In this paper, we will show an example where the match bottleneck for production systems [1, 3] is eased by utilizing rule-level parallelism. The example involves setting up a cafeteria for different functions and is indicative of the possibilities of performance improvement with PCLIPS [13]. A second example of a battle management expert system provides a perspective to real world applications in PCLIPS.

---

[1]This research partially supported by a grant from Honeywell and a grant from the Software section of the Florida High Technology and Research Council

[2]Also at Graduate Programs in Software, University of St. Thomas, 2115 Summit Ave PO 4314, St. Paul, MN 55105, bhbennett@stthomas.edu.

The rest of the paper consists of a description of PCLIPS, a section describing the knowledge bases (and parallelization approaches) of the examples and speed-up results from running them using PCLIPS, and a summary of experiences with parallel CLIPS.

THE PCLIPS SYSTEM

Based on experience with an early prototype, the design of the PCLIPS user interface models that of CLIPS as much as possible. A small extension to the syntax is used to allow the user to access working memory on each node, add/retract facts or rules to specific nodes, etc. For example, a load command with four processors allocated now takes the form: (0 1, load "cafe") and (, load "cafe"). The first command will load files cafe0 to node 0 and cafe1 to node 1, and the second command loads files cafe0, cafe1, cafe2, and cafe3 onto nodes 0, 1, 2 and 3. Other commands operate in the same way with (2, facts) bringing in the facts from node 2 and (3 7, rules) causing the rules from processors 3 and 7 to be displayed.

After rule firing is complete in PCLIPS, the amount of time spent by each node in the match, recognize, and act cycles is displayed. The amounts of time are given as percentages of the overall time, which is also displayed. Sequential timings are obtained from running PCLIPS on one node.

A complete version of CLIPS 5.1 enhanced with three parallel operations, xassert, xretract and mxsend, runs on each of the nodes and the host of an iPSC2 hypercube. The host node automatically configures each of the allocated nodes without user intervention when PCLIPS is invoked. The xassert command simply asserts a fact from one node to a remote node's working memory. For example, (xassert 3 (example fact)) makes its assertion into the working memory of node 3. The general form is (xassert node_number fact_to_assert). To retract a fact from a remote working memory use (xretract node_number fact_to_retract). Both operations build a message and cause it to be sent by the hypercube operating system. Neither command depends upon a specific message passing hardware or software mechanism.

Long messages can take less time to send than many short messages on Intel Hypercubes [2, 8] so mxsend() provides the user with the capability of asserting and/or retracting multiple facts into/from one processor to another processor. The syntax of the function is as follows: (mxsend node_numbers). Mxsend() needs a sequence of calls in order for it to work as desired. The first step in correctly building a message to be used by mxsend() is to call the function clear_fact(). The syntax for this function is as follows: (clear_fact). This function simply resets the buffer used by mxsend() to the '~' character. This character is necessary for a receiving processor to recognize the received message was sent by using mxsend(). The second step is to actually build the message to be sent. In order to do this, a sequence of calls to the function buildfact() should be performed. The syntax of buildfact() is as follows: (buildfact action fact). There are four possible values for the action variable. They are '0', '1', 'retract', and 'assert'. The '0' flag and 'retract' will both cause the building of a message to retract a fact (this is done by inserting a '$' character in the message buffer followed by fact), and '1' and 'assert' will both cause the building of a message to assert fact (this is done by inserting a '#' character in the message buffer followed by fact). If the following sequence of calls is performed, (buildfact assert Hello World) (buildfact retract PARCLIPS is fun) (buildfact assert Go Bulls!!!) (buildfact assert Save the Earth) the following string will be created: "~#Hello World$PARCLIPS is fun#Go Bulls!!!#Save the Earth" Finally, the function mxsend() can be called. Mxsend() will send the message built to the specified processors so that the message will be processed by the receiving processors. The call (mxsend 10 11 12), will cause the previously built message to be sent to processors 10, 11, and 12. The proper action is taken by the receiving processors who either assert or retract facts into/from their working memory.

Since PCLIPS is a research prototype, the user is free to use or misuse the parallel calls in any way he/she chooses. No safeguards are currently provided. On the other hand, the interface is simple and the calls straightforward. The question that comes to mind is whether they provide enough power to enable useful speed-ups on MIMD architectures. Our current work shows that they are suitable for obtaining useful speedups [13], if the knowledge base is parallelized in a careful and appropriate way.

Examples

In this section we show results from parallelizing a knowledge base and discuss a real application for parallel expert systems. All speedups are reported as the sequential time or time on one node divided by the time to process the same set of initial facts and obtain the same set of final facts in parallel (Sequential Time/Parallel Time).

Before discussing examples, we discuss a few guidelines for parallelizing rule bases that have become clear in the course of developing and testing PCLIPS.

Parallelizing Knowledge Bases

There are several approaches that have been taken to parallelizing knowledge bases [5, 7, 9, 11]. An important aspect is that the parallel results be equivalent to the serial results. Methods of explicit synchronization [11] do not seem feasible until communication times are significantly reduced on parallel machines. Hence, we have pursued serialization through rule base modification. This means that the rules in a parallel knowledge base generated under our paradigm are not necessarily syntactically the same as a set of sequential rules.

There are two approaches to parallelizing the rules of a specific sequential knowledge base. The first, and most usual one, is to partition the rules independent of the types of facts they will most likely be used with. In this approach, bottleneck rules that may need to be distributed to multiple processors must be searched for during a sequential trace of the knowledge based system's operation. Processors must be load balanced with an appropriate number of rules. All parallel actions must be inserted into the right hand side of the rules. All facts will be distributed to all nodes under this paradigm.

The second approach to parallelizing the knowledge base is to parallelize it based upon the rules and the expected type of facts. This approach is only feasible if a rule base may be expected to work with one type or set of facts (with the facts themselves changing) in most cases. This approach involves an analysis of the sequential performance of the knowledge based system with a specific set of facts and then a parallelization of the knowledge base for a set of processors. In the limited testing done in our work, this approach to parallelizing rules provides a greater speed-up.

Cafeteria

There are 93 rules in our version of the cafeteria knowledge base. The rules are grouped into contexts, where an example context involves setting a table. A rule and fact partitioning of the cafeteria knowledge base was done with the use of xassert and xretract and a speedup of 5.5 times was obtained using eight processors. The speedup obtained without using these functions was 6.47 times also using eight processors. Both speedups are less than linear but notice the decrease in speedup when using xassert and xretract. The decrease in speedup is here attributed to inter-processor communication. The time required to decode a message and assert it into working memory is between 1-2 msec [10]. The time used to obtain the above results includes the time required to transmit facts across to other nodes, retract/assert them into working memory, and do

the complete inferencing. A single message of 1K takes 1.1 msec to process [2]. Larger messages, however, take considerably more time to process, as shown by Boman and Roose [2]. Since, for these partitions, the messages sent across the nodes are larger than 1Kbyte (every node concatenates approximately 50 35-byte messages, making messages of 1.7 Kbytes that are sent using mxsend()), and all nodes transmit their messages to the same node (messages might have to wait on intermediate nodes and hence are blocked until memory on the destination node is available to receive the complete message [10, 2]). It is clear from the above that communication is the reason for the decrease in speedup.

The cafeteria knowledge base was also partitioned using 11 and 13 processors. A speedup of 11.5 was obtained using 11 processors, whereas the 13-processor partition produced a speedup of 22.85 times. A fact-based partitioning method was used to obtain both of these partitions. These speed-ups are clearly super-linear and occur because the match percentage of time is reduced in a non-linear fashion by this partitioning approach [13]. Due to space limitations, we will not explore this phenomenon further but refer the reader to our technical report [4].

Finally, several two-processor partitions of cafeteria were performed partitioning the rules only. A speedup of 2.035 times (65.99% matching, 21.11% acting) with two processors was obtained. In this case, rules were copied to each partition unmodified, causing the assertion of facts that are never used by the partition (since the asserted facts enable the firing of a context present in another partition). Partitioning the facts also, the speedup obtained was 2.06 (67.41% matching, 20.26% acting), which is only slightly higher than the speedup obtained when the facts were left intact. Notice that this result suggests that the number of extra unnecessary facts does not significantly affect the overall parallel execution time. A final two-processor partition was performed by modifying the rules left in each partition so that they assert only the context facts needed in the partition. A speedup of 2.13 times was obtained in this case.

Battle Management Expert System

The information fusion problem for battle management occurs when multiple, disparate sensor sources are feeding an intelligence center. This intelligence center is trying to produce timely, accurate and detailed information about both enemy and friendly forces in order for commanders to make effective battle management decisions. The challenge to the C^3I operation is to integrate information from multiple sources in order to produce a unified, coherent account of the tactical, operational or strategic situation.

There has recently been a vast proliferation of fixed and mobile, land- and air-based sensors using acoustic, infrared, radar and other sensor technologies. The result of this proliferation has made more work for the C^3I operation.

Sensors can vary in a variety of dimensions including:

- Coverage Area
- Temporal Characteristics of Coverage
- Field of View
- Angle of View
- Range
- Resolution
- Update Rate
- Detection Probability
- Modality of Imagery
- Degree of Complexity/Realism of Imagery
- Type of Target Information
- Temporal Characteristics of Reports

Each collection system, then, gives a specialized sampling of conditions to a particular level of detail, in specific locations, at a specific point in time, and with a particular level of accuracy. As a result, the analyst receives information that may be incompatible, fragmentary, time-disordered, and with gaps, inconsistencies and contradictions.

Honeywell's Combat Information Fusion Testbed (CIFT) has been developed to provide the hardware and software environment that can support development of tools powerful enough to assist intelligence analysts in correlating information from widely disparate sources. The current testbed capabilities were chosen for the context of handling three sensors: an airborne moving target indication (MTI) radar, a standoff signal intelligence (SIGINT) system, and an unmanned aerial vehicle (UAV) with a television camera payload. This correlation capability is fundamental for information fusion. By integrating Honeywell's proprietary real-time blackboard architecture (RTBA) with the proprietary spatial-temporal reasoning technique called topological representation (TR), the testbed has been able to perform the data association task. CIFT was developed and tested against a four-hour European scenario involving troop movement in a 40X60 km area that was observed by an MTI radar, a SIGINT system, and a UAV. We determined the target detections and circular error probabilities and time delay that these three systems would be expected to make. CIFT was found to operate effectively on this data, associating reports from the different sensors that had emanated from the same target.

CIFT was then implemented on the Intel iPSC-860 parallel processor [14] producing Parallel-CIFT (or Parallel-CIFT). This processor has eight parallel nodes. There are three major components of the CIFT system: Geographic/Scenario data, Blackboard Control Structures, Spatial/Temporal Reasoners.

Geographic/Scenario Data: These contain the bit maps of the map overlays and the scenario-specific operational and doctrinal data. The current scenario illustrates a Motorized Rifle Regiment in the Fulda area of eastern Germany mobilizing for a road march. This activity includes SIGINT, AUV (airborne unmanned vehicles with video camera payloads), and MTI (moving target indicator radar) sensor reports to a G2 intelligence workstation. The geographic data includes overlays for cities, primary and secondary roads, dense vegetation, and railroads.

Blackboard Control Structures: CLIPS provides the control and representation structures for the blackboard control architecture. Honeywell wrote data structures and fusion rules in the CLIPS format on a Sun workstation. These components were then parallelized and ported to the iPSC-860. Three demonstrations are available: one uses only one of the nodes on the parallel processor (this simulates a traditional serial computer for bench marking purposes), one uses two parallel nodes, and one uses four parallel nodes.

Spatial/Temporal Reasoning: The spatial/temporal reasoner for this system is built on a four-dimensional reasoner developed from Allen's temporal interval reasoning system. It defines the relations that can exist between time and space events and reasons from these primary relations.

This system represents a demonstration of concept of the Parallel CIFT system, a challenging problem in a challenging domain which effectively uses the Parallel CLIPS tool.

Current research efforts include:

- **Auto allocation of parallel components**—This work requires some basic and applied research. We propose using a nearest neighbor shear sort algorithm to dynamically allocate processing tasks across multiple processors. This will balance the load among the processors and ensure optimal performance.

- **Demonstrate P-CIFT and extensions on Paragon**—This requires three preliminary steps: 1) Port CLIPS (the forward-chaining inference engine, on which P-CIFT is built) to the Paragon, 2) Port P-CIFT to the Paragon, 3) Extensions developed to P-CIFT. See following points.

- **Addition of object-oriented data base (OODB) capabilities**—This should be easily completed with use of the CLIPS 6.0.

- **Development of a domain specific information fusion shell**—Common elements from a variety of information fusion applications (Honeywell currently has Army and Navy scenarios, with plans to extend into commercial domains, medical imaging, robotics, and electronic libraries specifically, in the next year) will be formalized and generalized for future use on other systems. This organic growth of generic components will assure the applicability, generality and usefulness.

- **Multi-hypothesis reasoning**—This will require integration of techniques for multiple hypothesis generation, maintenance, and testing. Previous related work [15] has demonstrated successful approaches in tasks with similar multiple assignment requirements. New research would be required to examine parallel implementation of these approaches. It is likely that a parallel approach could be much more efficient.

- **Quantification of performance results**—Past work has provided demonstrations of concept, but has provided no performance results.

SUMMARY

In this paper, we have discussed a parallel version of the CLIPS 5.1 expert system tool. The parallel tool has a simple interface that is a direct extension of the usual CLIPS interface for parallel use. The tool makes use of rule-level parallelism and has been tested on Intel Hypercubes. Examples of expert systems that may be parallelized have been shown. The major bottleneck involves developing effective and automated methods of parallelizing knowledge bases.

The cafeteria knowledge base example shows that good speed-up is possible from just rule-level parallelism. In fact, in the cases where both rule and fact partitioning can be done the speed-up is super-linear in this example. It appears the approach of rule-level parallelism holds significant promise for parallel expert system implementation on MIMD distributed memory computers.

The Parallel Combat Information Fusion Testbed represents a challenging real-world application of Parallel CLIPS technologies.

REFERENCES

[1] Newell, A., Gupta, A., and Forgy, C. "High-Speed Implementations of Rule-Based Systems," ACM TRANSACTIONS ON COMPUTER SYSTEMS, Vol. 7(2), 1989, pp. 119-146.

[2] Boman, L. and Roose, D. "Communication Benchmarks for the ipsc/2," PROCEEDINGS OF THE FIRST EUROPEAN WORKSHOP ON HYPERCUBE AND DISTRIBUTED COMPUTERS, Vol. 1, 1989, pp. 93-99.

[3] Gupta, A. PARALLELISM IN PRODUCTION SYSTEMS, Morgan-Kaufmann Publishers, Inc., Los Altos, CA, 1987.

[4] Hall, L.O. and Tello, I. "Parallel Clips and the Potential of Rule-Level Parallelism," ISL-94-71, Department of CSE, University of South Florida, Tampa, FL, 1994.

[5] Kuo, S and Moldovan, D. "Implementation of Multiple Rule Firing Production Systems On Hypercube," AAAI, Vol. 1, 1991, pp. 304-309.

[6] Miranker, D. P., Kuo, C., and Browne, J.C. "Parallelizing Transformations for a Concurrent Rule Execution Language," TR-89-30, University Of Texas, Austin, 1989.

[7] Neiman, D.E. "Parallel ops5 User's Manual and Technical Report," Department of Computer Science and Information Science, University of Massachusetts, 1991.

[8] Nugent, S.F. "The Ipsc/2 Direct-Connect Communications Technology," COMMUNICATIONS OF THE ACM, Vol. 1, 1988, pp. 51-60.

[9] Oflazer, K., "Partitioning in Parallel Processing of Production Systems," PROCEEDINGS OF THE 1984 CONFERENCE ON PARALLEL PROCESSING, 1984, pp. 92-100.

[10] Prasad, L. "Parallelization of Expert Systems in the Forward Chaining Mode in the Intel Hypercube," MS Thesis, Department of CSE, University of South Florida, Tampa, FL, 1992.

[11] Schmolze, J.G. "Guaranteeing Serializable Results in Synchronous Parallel Production Systems," JOURNAL OF PARALLEL AND DISTRIBUTED COMPUTING, Vol. 13(4), 1991, pp. 348-365.

[12] Schmolze, J.G. and Goel, S. "A Parallel Asynchronous Distributed Production System," PROCEEDINGS OF AAAI-90, 1990, pp. 65-71.

[13] Tello, I. "Automatic Partitioning of Expert Systems for Parallel Execution on an Intel Hypercube," MS Thesis, Department of CSE, University of South Florida, Tampa, FL, 1994.

[14] Bennett, B. H. "The Parallel Combat Information Fusion Testbed," Honeywell High Performance Computing Workshop, Clearwater, FL, December 1992.

[15] Bennett, B.H. "A Problem-solving Approach to Localization," PhD Thesis, University of Minnesota, February 1992.

Using CLIPS to Represent Knowledge in a VR Simulation

Mark Engelberg
LinCom Corporation
mle@gothamcity.jsc.nasa.gov

September 13, 1994

Abstract

Virtual reality (VR) is an exciting use of advanced hardware and software techonologies to achieve an immersive simulation. Until recently, the majority of virtual environments were merely "fly-throughs" in which a user could freely explore a 3-dimensional world or a visualized dataset. Now that the underlying technologies are reaching a level of maturity, programmers are seeking ways to increase the complexity and interactivity of immersive simulations. In most cases, interactivity in a virtual environment can be specified in the form "whenever such-and-such happens to object X, it reacts in the following manner." CLIPS and COOL provide a simple and elegant framework for representing this knowledge-base in an efficient manner that can be extended incrementally. The complexity of a detailed simulation becomes more manageable when the control flow is governed by CLIPS' rule-based inference engine as opposed to by traditional procedural mechanisms. Examples in this paper will illustrate an effective way to represent VR information in CLIPS, and to tie this knowledge base to the input and output C routines of a typical virtual environment.

1 Background Information

1.1 Virtual Reality

A virtual experience, or more precisely, a sense of immersion in a computer simulation, can be achieved with the use of specialized input/output devices. The head-mounted display (HMD) is perhaps the interface that most characterizes virtual reality. Two small screens, mounted close to the user's eyes, block out the real world, and provide the user with a three-dimensional view of the computer model. Many HMDs are mounted in helmets which also contain stereo headphones, so as to create the illusion of aural, as well as visual immersion in the virtual environment. Tracking technologies permit the computer to read the position and angle of the user's head, and the scene is recalculated accordingly (ideally at a rate of thirty times a second or faster). [1]

There are many types of hardware devices which allow a user to interact with a virtual environment. At a minimum, the user must be able to navigate through the envrionment. The ability to perform actions or select objects in the environment is also critical to making a virtual environment truly interactive. One popular input device is the DataGlove which enables the user to specify actions and objects through gestures. Joysticks and several variants are also popular navigational devices.

1.1.1 Training

Virtual reality promises to have a tremendous impact on the way that training is done, particularly in areas where hands-on training is costly or dangerous. Training for surgery, space missions, and combat all fall into this category; these fields have already benefitted from existing simulation technologies [2]. As Joseph Psotka explains, "Virtual reality offers training as experience" [3, p. 96].

1.1.2 Current Obstacles

VR hardware is progressing at an astonishing rate. The price of HMDs and graphics workstations continues to fall as the capabilities of the equipment increase. Recent surveys of the literature have concluded that the biggest

obstacle right now in creating complex virtual environments is the software tools. One study said that creating virtual environments was "much too hard, and it took too much handcrafting and special-casing due to low-level tools" [5, p. 6].

VR developers have suffered from a lack of focus on providing interactivity and intelligence in virtual environments. Researchers have been "most concerned with hardware, device drivers and low-level support libraries, and human factors and perception" [5, p. 6]. As a result,

> Additional research is needed to blend multimodal display, multisensory output, multimodal data input, the ability to abstract and expound (intelligent agent), and the ability to incorporate human intelligence to improve simulations of artifical environments. Also lacking are the theoretical and engineering methodologies generally related to software engineering and software engineering environments for computer-aided virtual world design. [4, p. 10]

1.2 CLIPS

VR programmers need a high-level interaction language that is object-oriented because "virtual environments have potentially many independent but interacting objects with complex behavior that must be simulated" [5, p. 6]. But unlike typical object-oriented systems, there must be "objects that have time-varying behavior, such as being able to execute Newtonian physics or *systems of rules*" [5, p. 7].

CLIPS 6.0 can fill this need for a high-level tool to program the interactions of a virtual environment. COOL provides the object-oriented approach to managing the large number of independent objects in complex virtual environments, and CLIPS 6.0 provides the ability to construct rules which rely on pattern-matching on these objects.

CLIPS is a particularly attractive option for VR training applications because many knowledge-bases for training are already implemented using CLIPS. If the knowledge-base about how different objects act and interact in a virtual environment is implemented in the same language as the knowledge-base containing expert knowledge about how to solve tasks within the environment, then needless programming effort can be saved.

2 Linking CLIPS with a Low-level VR Library

2.1 Data Structures

Every VR library must use some sort of data structure in order to store all relevant positional and rotational information of each object in a virtual environment. These structures are often called *nodes* and nodes are often linked hierarchically in a tree structure known as a *scene*. In order to write CLIPS rules about virtual objects, it is necessary to load all the scene information into COOL.

The following NODE class has slots for a parent node, children nodes, x-, y-, and z-coordinates, and rotational angles about the x-, y-, and z-axis.

```
(defclass NODE
  (is-a USER)
  (role concrete)

  (slot node-index (visibility public)
        (create-accessor read-write) (type INTEGER))

  (slot parent (visibility public)
        (create-accessor read-write) (type INSTANCE-NAME))
  (multislot children (visibility public)
             (create-accessor read-write) (type INSTANCE-NAME))

  (slot x (visibility public) (create-accessor read) (type FLOAT))
  (slot y (visibility public) (create-accessor read) (type FLOAT))
  (slot z (visibility public) (create-accessor read) (type FLOAT))
  (slot xrot (visibility public) (create-accessor read) (type FLOAT))
  (slot yrot (visibility public) (create-accessor read) (type FLOAT))
  (slot zrot (visibility public) (create-accessor read) (type FLOAT)))
```

It is a trivial matter to write a converter which generates NODE instances from a scene file. For example, a typical scene might convert to:

```
(definstances tree
(BODY of NODE)
(HEAD of NODE)
```

```
; and so on
)

(modify-instance [BODY]
        (x 800.000000)
        (y -50.000000)
        (z 280.000000)
        (xrot 180.000000)
        (yrot 0.000000)
        (zrot 180.000000)
        (parent [REF])
        (children [HEAD] [Rpalm] [Chair]))
; and so on
```

2.2 Linking the Databases

Note that the NODE class has no default write accessors associated with slots x, y, z, xrot, yrot, or zrot. Throughout the VR simulation, the NODE instances must always reflect the values of the node structures stored in the VR library, and vice versa. To do this, the default write accessors are replaced by accessors which call a user-defined function to write the value to the corresponding VR data structure immediately after writing the value to the slot. Similarly, all of the VR functions must be slightly modified so that any change to a scene node is automatically transmitted to the slots in its corresponding NODE instance.

The node-index slot of the NODE class is used to give each instance a unique identifying integer which serves as an index into the C array of VR node structures. This helps establish the one-to-one correspondence between the two databases.

2.3 Motion Interaction

A one-to-one correspondence between the database used internally by the VR library and COOL objects permits many types of interactions to be easily programmed from within CLIPS, particularly those dealing with motion. The following example illustrates how CLIPS can pattern-match and change slots, thus affecting the VR simulation.

```
(defrule rabbit-scared-of-blue-carrot
  (object (name [RABBIT]) (x ?r))
  (object (name [CARROT]) (x ?c&:(< ?c (+ ?r 5))&:(> ?c (- ?r 5))))
  =>
  (send [RABBIT] put-x (- ?r 30)))
```

Assuming the rabbit and blue carrot can only move along the x-axis, this rule can be paraphrased as "whenever the blue carrot gets within 5 inches of the rabbit, the rabbit runs 30 inches away."

2.4 The Simulation Loop

Most interactivity in virtual environments can be almost entirely specified by rules analogous the above example. A simulation then consists of the following cycle repeated over and over:

1. The low-level VR function which reads the input devices is invoked.
2. The new data is automatically passed to the corresponding nodes in COOL, as described in Section 2.1.
3. CLIPS is run, and any rules which were activated by the new data are executed.
4. The execution of these rules in turn triggers other rules in a domino-like effect until all relevant rules for this cycle have been fired.
5. The VR library renders the new scene from its internal database (which reflects all the new changes caused by CLIPS rules), and outputs this image to the user's HMD.

While the low-level routines still provide the means for reading the input and generating the output of VR, the heart of the simulation loop is the execution of the CLIPS rule-base. This provides an elegant means for incrementally increasing the complexity of the simulation; best of all, CLIPS' use of the Rete algorithm means that only relevant rules will be considered on each cycle of execution. This can be a tremendous advantage in complex simulations.

2.5 More VR functions

There are some VR interactions that cannot be accomplished solely through motion. Fortunately, CLIPS provides the ability to create user-defined functions. This allows the programmer to invoke external functions from within CLIPS. User-defined CLIPS functions can be provided for most of the useful functions in a VR function library so that the programmer can use these functions on the right-hand side of interaction rules. Some useful VR functions include:

make-invisible Takes the **node-index** of an instance as an argument.

change-color Requires the **node-index** of an instance and three floats specifying the red, green, and blue characteristics of the desired color.

test-collision Takes two **node-index** integers and determines whether their corresponding objects are in contact in the simulation.

play-sound Takes an integer which corresponds to a soundfile (this correspondence is established in another index file of soundfiles).

The **play-sound** function takes an integer as an argument, instead of a string or symbol, because there is slightly more overhead in processing and looking up the structures associated with strings, and speed is crucial in a virtual reality application. Similarly, all user-defined functions should receive the integer value stored in an instance's **node-index** slot, instead of the instance-name, for speed purposes.

It is also possible to create a library of useful deffunctions which do many common calculations completely within CLIPS. For example, a distance function, which takes two instance-names and returns the distance between their corresponding objects, can be written as follows:

```
(deffunction distance (?n1 ?n2)
  (sqrt (+ (** (- (send ?n2 get-x) (send ?n1 get-x)) 2)
           (** (- (send ?n2 get-y) (send ?n1 get-y)) 2)
           (** (- (send ?n2 get-z) (send ?n1 get-z)) 2))))
```

3 Layers upon Layers

Once basic VR functionality is added to CLIPS, virtual environments can be organized in CLIPS according to familiar object-oriented and rule-based design principles. For example, a RABBIT class could be defined, and the example rules could be modified to pattern match on the is-a field instead of the name field. If this were done, any RABBIT instance would automatically derive the appropriate behavior. Once a library of classes is developed and an appropriate knowledge-base to go with it, creating a sophisticated virtual environment is merely a matter of instantiating these classes accordingly.

Consider this final example, illustrating points from Sections 2.5 and 3.

```
(defrule shy-rabbit-behavior
  ?rabbit <- (object (is-a RABBIT) (personality shy)
                     (x ?) (y ?) (z ?))
  ?hand <- (object (is-a HAND) (x ?) (y ?) (z ?))
  =>
  (if (< (distance ?rabbit ?hand) 100) then
    (bind ?rabbit-index (send ?rabbit get-node-index))
    (if (evenp (random)) then
      (change-color ?rabbit-index 1 0 0)  ; rabbit blushes
      else
      (make-invisible ?rabbit-index))))
```

This rule becomes relevant only if there is a shy rabbit and a hand in the simulation. If so, shy-rabbit-behavior is activated whenever the hand or the rabbit moves. If the hand gets close to the rabbit, there is a 50% chance that the rabbit will blush, and a 50% chance that the rabbit will completely vanish.

4 Conclusion

CLIPS has been successfully enabled with virtual reality programming capabilities, using the methodologies described in this paper. The two test users of this approach have found CLIPS to be a simpler, more natural paradigm for programming virtual reality interactions than the standard approach of managing and invoking VR functions directly in the C language. Hopefully,

by using high-level languages like CLIPS in the future, more VR programmers will be freed from their current constraints of worrying about low-level details, and get on with what really matters — creating complex, intelligent, interactive environments.

References

[1] Ben Delaney. State of the art VR technology circa 1994. *New Media*, 4(8):44–45, August 1994.

[2] Ben Delaney. Virtual reality goes to work. *New Media*, 4(8):40–48, August 1994.

[3] Joseph Psotka. Synthetic environments for training. In *Second Annual Synthetic Environments Conference*, pages 79–107. Technical Marketing Society of America, March 1994.

[4] U.S. Army Training and Doctrine Command and U.S. Army Research Office. *Executive Summary — Virtual Reality/Synthetic Environments in Army Training*, October 1992.

[5] Andries van Dam. VR as a forcing function: Software implications of a new paradigm. In *IEEE 1993 Symposium on Research Frontiers in Virtual Reality*, pages 5–8, Los Alamitos, California, October 1993. IEEE Computer Society Technical Committee on Computer Graphics, IEEE Computer Society Press.

N95-19766

Reflexive Reasoning for Distributed Real-Time Systems

David Goldstein
Computer Science Department
North Carolina Agricultural & TechnicalState University
Greeensboro, NC 27411
USA
Voice: (910) 334-7245
Fax: (910) 334-7244
goldstn@garfield.ncat.edu

Abstract

This paper discusses the implementation and use of reflexive reasoning in real-time, distributed knowledge-based applications. Recently there has been a great deal of interest in agent-oriented systems. Implementing such systems implies a mechanism for sharing knowledge, goals and other state information among the agents. Our techniques facilitate an agent examining both state information about other agents and the parameters of the knowledge-based system shell implementing its reasoning algorithms. The shell implementing the reasoning is the Distributed Artificial Intelligence Toolkit, which is a derivative of CLIPS.

Introduction

There has been a great deal of recent interest in multi-agent systems largely due to the increasing cost-effectiveness of utilizing distributed systems; in just the national CLIPS conference six papers appear which seem to discuss developing multi-agent systems. Further, although we strenuously try to avoid incorporating domain-specific knowledge in systems, real-time applications have an obvious need to understand the relationships between their processing requirements, available system resources, deadlines which must be met, and their environment. Hence, our programming methodology has been that individual agents need not necessarily be cognizant of any system information, but rather can communicate their own informational requirements, can sense their state in the system, and modify the internal processing parameters of the system (e.g., for load balancing) as the application demands. We allow the agents to sense and affect their processing environment so that they can intelligently reason about and affect the execution of applications.

Implementation of Reflexive Reasoning

We have previously documented the characteristics of our tool, the Distributed Artificial Intelligence Toolkit [1][2]. The tool provides extensions to CLIPS for fault-tolerant, distributed, real-time reasoning. Many of these characteristics are controllable by individual reasoning agents so that when insufficient processing time is available, for example, processor fault-tolerance may need to sacrificed. The control of such characteristics is provided numerous predicates. Correspondingly, the agents can sense the current settings of the environment through a state description of the inference engine contained in the fact base (and which can be pattern-matched).

Calls to such predicates, as well as numerous 'C' functions implemented to provide additional functionality, were used to implement the Agent Manipulation Language (AML). AML (Table 1) provides the functionality to manipulate, assign tasks to, and teach agents. The functions used to implement AML include those providing fault-tolerance, for transmitting facts, template and objects, and those mimicking

user-interface functionality; the data assistants of our architecture(Figure 1) actually interface to the user, evoking functionality from the reasoning agents [1].

build_agent(*processor*)	Creates an agent
teach_agent(*agent name, set of rules as text*)	Sends a ruleset to agent for reasoning
agent_unlearn(*agent name, set of rules as text*)	Causes agent to remove ruleset from its processing
agent_learn(*filename*)	Have agent read a rulebase from *filename*
die(*agent name*)	Kill the agent
wait(*agent name*)	Have the agent suspend reasoning
continue(*agent name*)	Have the agent continue reasoning

Table 1: Agent Manipulation Language (AML)

The last big issue is how the environment is sensed and affected at a low level. These processes are accomplished by intercepting and interpreting the individual elements of facts and templates before they are actually asserted. This kind of functionality allows agents to be minimally required to affect other agents; agents can know and affect each other (on the same or other machines) as much or as little as they desire. Formally proprietary information, this is now being divulged because implementing the code for the parsing of such information has been deemed too difficult for students, even graduate students, to maintain.

Reflexive Reasoning in Distributed Real-time Systems

Consider a real-time robotics application. The application consists of path planners, task planners, sensor and effector managers, motion control modules, etc. For the planning modules we typically would want to employ fault-tolerance, but for many of the other modules we would want very fast update rates. Hence, we might initially turn off fault-tolerance for, turn on interruptable reasoning for, and reduce the granularity of reasoning for the machines controllers, sensor managers, motion control modules, etc.

Certain planners and motion control modules would probably would probably require more resources than others. Modules noting short times between actual completion of tasks and the tasks' actual deadlines can evoke operating system functionality, via the data assistants, to determine processors with "excess" computing power. The over burdened agents could then create new agents, advise them to learn a set of rules, and off-load some of their work to newly instantiated agents. Hence, as the system executes, overworked agents could instigate load balancing.

Agents can also use reflexive reasoning in less subtle manners; any agent can request to see the fact and object base of any other agent. Agents can also request to know the goals of interest of other agents (or rather, which agents are interested in what goals). Hence, agents can also reason about the reason about the reasoning being performed by other agents.

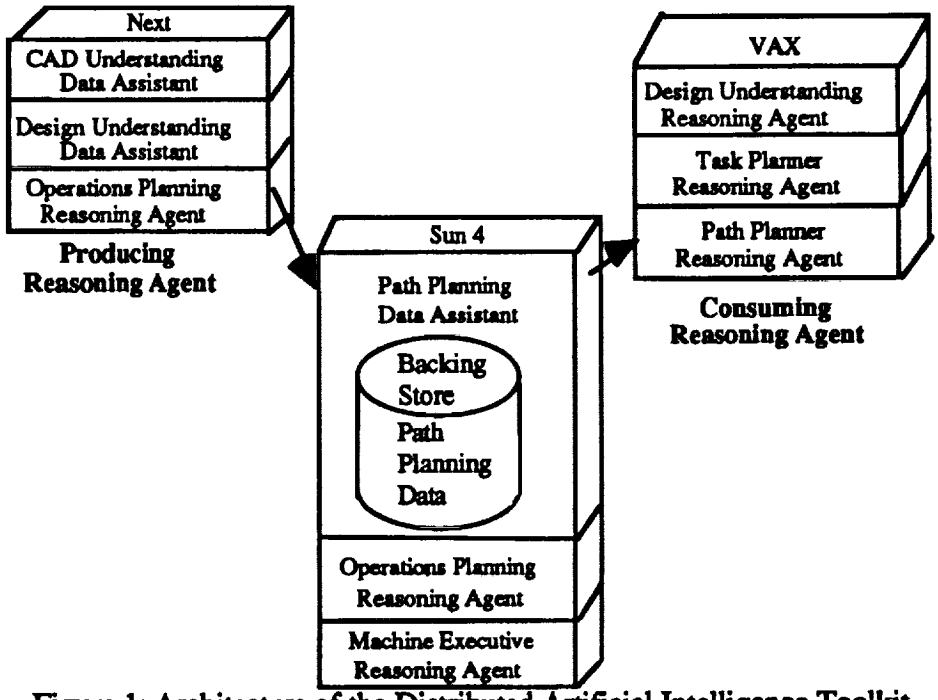


Figure 1: Architecture of the Distributed Artificial Intelligence Toolkit

Conclusion

We have briefly discussed implementing and using reflexive reasoning in distributed, real-time applications. Reflexive reasoning provides reasoning agents in distributed systems to analyze and modify the reasoning processes of other agents. We have found reflexive reasoning an effective tool for facilitating control of real-time, multi-agent systems. Our implementation of reflexive reasoning has been hierarchical, building up an agent manipulation language from predicates describing and affecting the reasoning process. These predicates have been, in turn, implemented from low-level functions written in 'C'.

References

1. Goldstein, David, "The Distributed Artificial Intelligence Toolkit", AI EXPERT, Miller-Freeman Publishing, January, 1994, pp 30-34.

2. Goldstein, David, "Extensions to the Parallel Real-time Artificial Intelligence System (PRAIS) for Fault-tolerant Heterogeneous Cycle-stealing Reasoning", in Proceedings of the 2nd Annual CLIPS ('C' Language Integrated Production System) Conference, NASA Johnson, Houston, September 1991.

omit

Session 7B: Diagnostic and Blackboard Extensions

Session Chair: Terry Feagin

PalymSys™ - An Extended Version of CLIPS for Construction and Reasoning Using Blackboards

Travis Bryson Dan Ballard
Reticular Systems, Inc.
4715 Viewridge Ave. #200
San Diego, CA 92123

Abstract

This paper describes PalymSys™ -- an extended version of the CLIPS language that is designed to facilitate the implementation of blackboard systems. The paper first describes the general characteristics of blackboards and shows how a control blackboard architecture can be used by AI systems to examine their own behavior and adapt to real-time problem-solving situations by striking a balance between domain and control reasoning. The paper then describes the use of PalymSys™ in the development of a situation assessment subsystem for use aboard Army helicopters. This system performs real-time inferencing about the current battlefield situation using multiple domain blackboards as well as a control blackboard. A description of the control and domain blackboards and their implementation is presented. The paper also describes modifications made to the standard CLIPS 6.02 language in PalymSys™ 2.0. These include: 1) A dynamic Dempster-Shafer belief network whose structure is completely specifiable at run-time in the consequent of a PalymSys™ rule, 2) Extension of the *run* command including a continuous run feature that enables the system to run even when the agenda is empty, and 3) A built-in communications link that uses shared memory to communicate with other independent processes.

Introduction

This paper describes the extensions made to the CLIPS 6.02 language during the design and implementation of a Situation Assessment (SA) expert system for use aboard Army helicopters. An SA system uses data gathered from external environmental sensors, intelligence updates, and pre-mission intelligence to monitor and describe the external environment. An SA system searches for external entities of interest (EEOI), recognizes those EEOIs, and then infers high-level attributes about them. An EEOI is anything that has the potential for affecting the planned rotorcraft mission. EEOIs are primarily (although not necessarily) enemy forces. In order for the system to perform the inferences necessary to develop an assessment of the current situation, it must utilize extensive knowledge about the EEOIs including knowledge about their doctrine, capabilities, probable mission objectives, intentions, plans, and goals. All of these elements combine to form a complete situation description. For a thorough description of the domain problem see [1].

The SA system has been implemented in an extended version of CLIPS called PalymSys™. The SA system implementation makes use of two domain blackboards - current assessment and predicted assessment, as well as a control blackboard for overall control of the system. PalymSys™ provides a reasoning under uncertainty mechanism that handles contradictory and partially contradictory hypotheses and allows multiple hypotheses to coexist. A continuous run option has been added that allows the system to run even when the agenda is empty. Continuous run enables the system to wait for new environmental state data to be provided by the system's sensor subsystems. As new data becomes available, additional reasoning is then performed.

Blackboards

The SA system uses a blackboard architecture as a paradigm for solving the situation assessment problem. The blackboard architecture approach to problem solving has been a popular model for expert system design since the development of the Hearsay-II speech understanding program in the 1970s. It also serves as a framework for the blackboard control architecture - an extension of the blackboard architecture - which is the method of control used in the SA system. The blackboard model for problem solving consists of three primary elements [2, 3]:

Knowledge Sources: The knowledge necessary to solve the problem is partitioned into separate and independent knowledge sources. The independence of knowledge sources means that major modifications to the system should not be necessary when more rules are added to the sys-

tem. In CLIPS and PalymSys™, knowledge sources take the form of rules.

Blackboard Data Structure: A global data structure where knowledge that has been brought to bear on the problem is stored. The blackboard represents the current state of the problem solution. The system attempts to combine and extend partial solutions that span a portion of the blackboard into a complete problem solution. Communication between knowledge sources takes place solely via the blackboard. In CLIPS, a blackboard data structure can be represented by objects that encapsulate the knowledge at each level. The knowledge is contributed by the consequent of rules whose antecedent has been satisfied.

Control: Each of the knowledge sources opportunistically contributes to the overall problem solution. Each knowledge source is responsible for knowing the conditions under which it will be able to contribute to the problem solution. In CLIPS, this means deciding which rule or set of rules should fire next given the current state of the blackboards. Our method for achieving this is the use of the control blackboard architecture. The control blackboard is an extension of the traditional blackboard architecture and will be discussed in detail later in this paper.

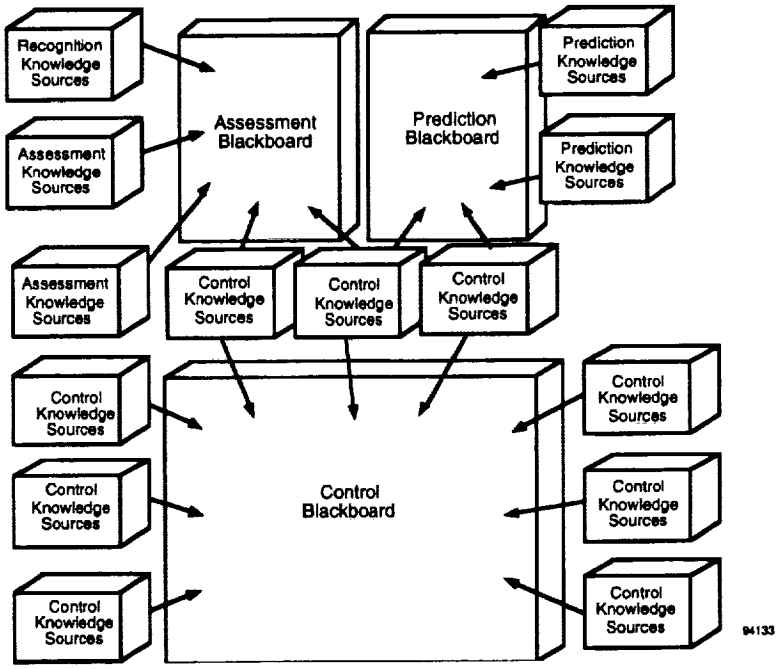


Figure 1. The Situation Assessment System Architecture

The SA system uses three concurrently executing blackboards for developing a problem solution. These are a *prediction* blackboard, an *assessment* blackboard, and a *control* blackboard. Each blackboard provides storage for the problem solution state data. The assessment blackboard contains the current situation description and is primarily concerned with the current intentions, capabilities, and commitments of EEOIs. The prediction blackboard contains predictions for EEOI behavior and the predicted situation description. The control blackboard contains the knowledge that manages and prioritizes all of the rules and provides for overall control of system problem-solving behavior.

Designing the SA Assessment Blackboard

The blackboard model provides only a general model for problem solving. It falls far short of an engineering specification for actually developing a complete blackboard system in CLIPS. However, this general model does provide significant insight in how to implement complex knowledge-based systems. The first step in designing a blackboard for a given domain problem is to subdivide the problem into discrete subproblems. Each subproblem represents roughly an independent

area of expertise. The subproblems are then organized into a hierarchy of levels from least to most abstract. Correctly identifying the problem hierarchically is crucial and will often be the primary factor that determines the effectiveness of the problem-solving system (or whether the problem can be solved at all). Blackboards sometimes have multiple blackboard panels, each with their own set of levels. That is, the solution space can be segmented into semi-independent partitions.

The knowledge sources used by the system are CLIPS rules that have access to the information on the blackboard. Communication and interaction among rules is solely via the blackboard data structure. Even knowledge sources on the same level must share information through the blackboard. Encoded within each knowledge source are the conditions under which it can contribute to the problem solution.

Figure 2 is an illustration of the assessment blackboard in the SA system. The assessment blackboard is divided into a seven-tiered hierarchy. These levels are concerned with developing an environmental state description, characterizing an EEOI, interpreting EEOI plans, roles, and intents and developing a summary description of the overall situation. Each level of the assessment blackboard is a *part-of* hierarchy that represents a portion of the situation assessment solution for a particular EEOI. There is a gradual abstraction of the problem as higher levels on the blackboard are reached. Information (properties) of objects on one level serve as inputs to a set of rules which, in turn, place new information on the same or adjacent levels. During the problem-solving process, more advanced hypotheses and inferences are placed at higher levels of the blackboard.

The blackboard architecture provides a model for the overall problem-solving (inferencing) process. This model is used to structure the problem domain and identifies the knowledge sources needed to solve the problem. While knowledge sources are independent and each contributes to a partial solution, each knowledge source must be designed to fit into the high-level problem-solving blackboard hierarchy created by the system designer.

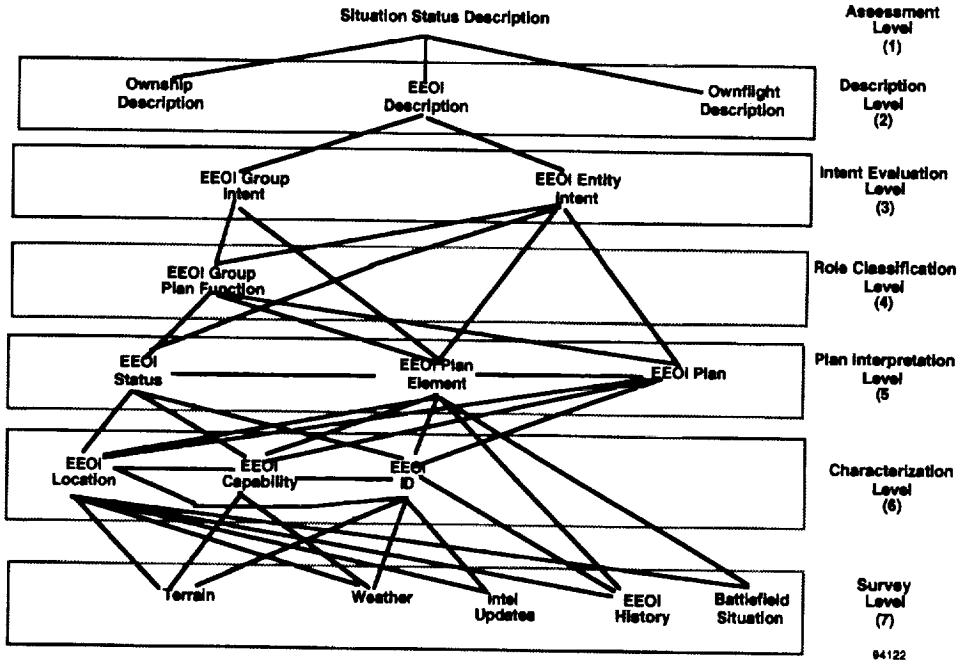


Figure 2. The Assessment Blackboard

Using CLIPS Objects as the Blackboard

Information on the assessment blackboard is represented as CLIPS objects. Figure 3 shows the object representation for knowledge in one of the SA modules. This figure shows the structure in the global plan function (GPF) module at the Role Classification level of the blackboard hierar-

chy.

A multi-agent plan hypothesis object is created by the system to represent the high-level multi-agent plan of a group of EEOIs where each has the same high-level mission objective. Multi-agent hypotheses and their associated entity plans are stored as objects on the blackboard. An entity plan object contains the sequence of plan elements (activities) that a particular EEOI must actually perform within the context of the associated multi-agent hypothesis. For instance, an EEOI's multi-agent plan might be to capture a refueling depot. A number of EEOIs will be needed to achieve this objective including a security force, a main attack force, and surveillance for the attacking force. The EEOI's entity plan might then be *surveillance* for the attacking force. The multi-agent hypothesis object called *capture refueling depot* encapsulates information local to the role classification level like formation information, hypothesized locations, and typical vehicle types. Other objects can access this information only through the multi-agent hypothesis object's defmessage handlers.

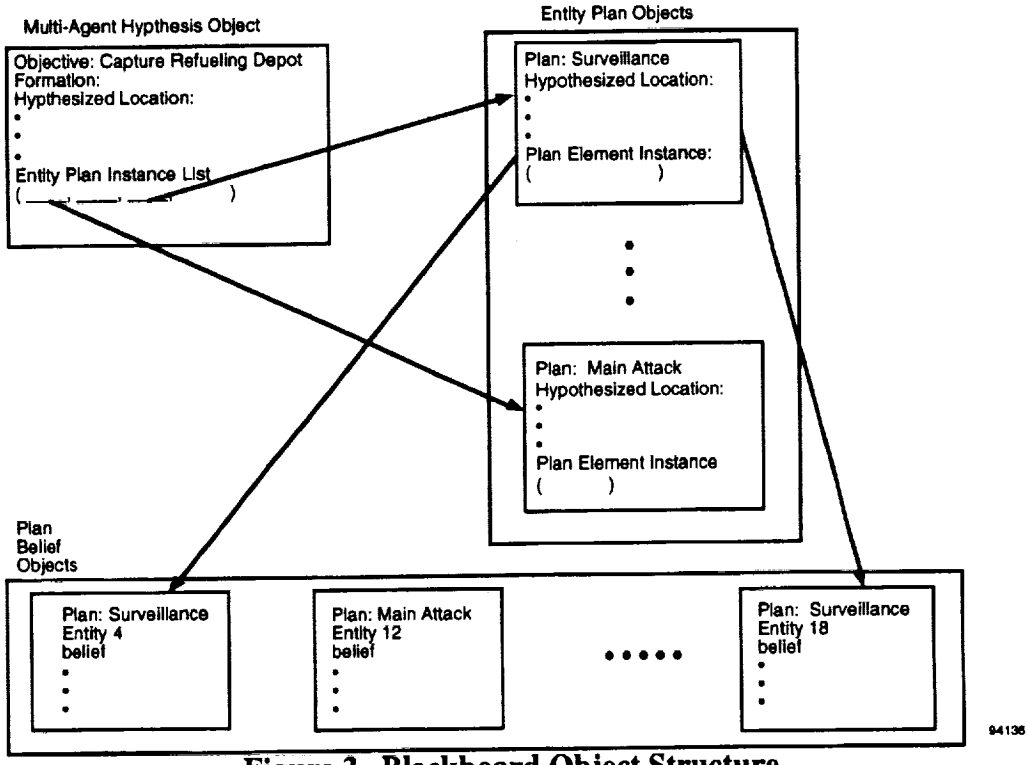


Figure 3. Blackboard Object Structure

The *capture refueling depot* object has a multislot field that contains the list of entity plan instances necessary to carry out its objective. Objects at different blackboard levels which are permanently linked are connected via instance lists. The entity plan instances, in turn, contain the lists of plan elements necessary to carry out the entity-level plan. The plan element lists are encapsulated within the entity plan objects. When the need to do so arises, entity plan objects will search for plan belief objects that correspond to their plan via pattern matching. They search for plan belief objects instead of parsing a pre-defined list because the links in this case are not permanent. The plan belief objects are entity specific and store the degree of belief in which the system believes that a particular EEOI is performing a particular plan. An EEOI may change entity plans or the system may gather evidence that leads it to believe the EEOI is actually performing a different plan. Thus the links between entity plan objects and plan belief objects will change over time.

CLIPS Objects are an ideal data structure for blackboard implementation because they offer encapsulation and easy processing of lists. Recall that the essence of the blackboard approach is

that higher levels synthesize the knowledge from the lower levels. The objects at one blackboard level will typically need access to a list of conclusions from the preceding levels. In our approach, objects are always looking down the blackboard, asking other objects for only the information they need. The knowledge at each blackboard level is well encapsulated. Knowledge sources can thus be formulated generically.

The Control Blackboard Architecture

At each point in the problem solving process there are typically a number of knowledge sources that can contribute to the problem solution. Every intelligent system must solve the control problem: i.e., determine which knowledge source should next be brought to bear on the problem solution. In order to solve the control problem it is necessary that control decision making be viewed as a separate problem-solving task. The system must plan problem-solving actions using strategies and heuristics that will help it solve the control problem while balancing efficiency and correctness. The system must become aware of how it solves problems and intelligently guide the problem-solving process.

Explicitly solving the control problem involves providing a body of meta-level (heuristic) knowledge about the domain that is used to guide the control planning process [4]. With such knowledge, the system can reason explicitly about control because the system has access to all of the knowledge that influences control decisions. Meta-level rules then choose domain rules or sets of domain rules that are most appropriate for the current problem-solving situation.

Control knowledge sources interact solely via the control blackboard. The control blackboard is where control knowledge sources post all currently relevant meta-level system knowledge. Partial and complete control plans are stored here. The system also posts high-level control heuristics and problem-solving strategies on the control blackboard.

A well designed control mechanism can make sophisticated meta-level decisions about the problem-solving process. It will seek to make desirable actions more feasible and feasible actions more desirable. It will make plans to seek out important obtainable information when that information is missing. The control mechanism must carefully balance the time spent solving control problems with time spent carrying out domain tasks. It must be aware of how it is solving domain problems and change problem-solving methods to match the situation requirements.

The Control Problem in SA

The SA system is a real-time system that must perform complex inferences within very demanding time constraints. Control is critical in a real-time system because by definition problems must be solved before a deadline. The SA system has a set of meta-level control rules that interact via the control blackboard. The control rule set uses heuristics to evaluate situation characteristics which are used to choose one or more problems from a pool of several competing domain problems. Once the important domain problems are chosen, an efficient control plan is constructed to solve them. A control plan consists of a series of rule groups, or modules, which will be sequentially accessed by the system. A message to work with a specific entity is frequently sent along with the control plan.

Control planning is a way of incorporating efficiency into the system. The meta-level priority criteria do not have to be recalculated every cycle while the system is following an established control plan. The system balances the degree of commitment to the execution of control plans with a variable sensitivity to run-time conditions [5]. The degree of commitment is a function of the uncertainty about the helicopter's actual environment. If the situation is dangerous, the system will lower its commitment to the control plan and heighten its sensitivity to run-time conditions (i.e., incoming sensor data).

Figure 4 is a diagram of the control blackboard used in SA. As in the assessment and prediction blackboards, a multilevel hierarchical blackboard structure is used. The *policy* level is where

global focusing decisions are stored. These are heuristically generated at run-time depending on the state of the problem-solving situation. The *problem* level is where domain problems presented to the system are placed. The *strategy* level is where strategies (high-level general descriptions of sequences of actions) are stored. Generating a strategy is the first step in making a plan. Strategy decisions are posted at the strategy level for all accepted problems. A strategy decision is nothing more than a constraint on future actions. The *focus* level is equivalent to the strategy level but is populated by more specific strategies called foci. The *action* level represents the actual sequence of actions chosen by the system to solve the problem [6]. The action level is implemented by the CLIPS agenda mechanism.

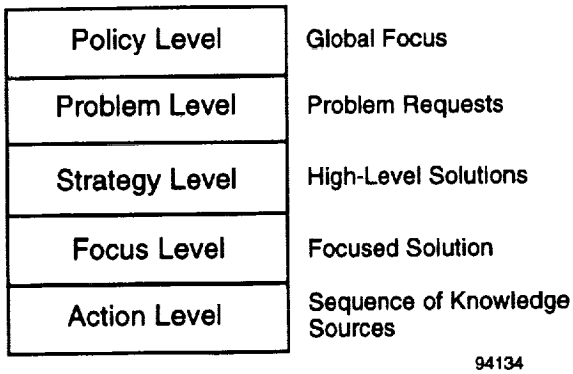


Figure 4. Control Blackboard Hierarchy

Meta Control Plans

The knowledge represented on each of the control blackboard levels increases in abstraction from the bottom level to the top level. However, control is a top-down inferencing process. Unlike the domain blackboards, knowledge at the higher blackboard levels serves as input for the knowledge sources at the lower control blackboard levels. A meta control plan solves the control problem for the control part of the SA system. A meta control plan object is constructed at system start-up that contains the following sequence of phases:

- check for new data
- post foci at policy level
- remove problems (if appropriate)
- request problems
- accept problems
- prioritize problems
- choose problems
- formulate strategy
- formulate plan
- execute a control plan
- record any plan completions and perform system maintenance

The control rules are partitioned by phase. There is a rule set for accepting problems, prioritizing problems, etc. The early phases deal with the higher levels on the control blackboard. The system will cycle through the meta control plan sequentially unless a message is passed from one module to another.

One situation in which message passing occurs is when the system suspects that a new domain problem should be reevaluated in light of the particular domain problem that has been chosen. For instance, when the system is making a plan to do a situation assessment, it will reconsider data that has not yet been integrated into the system. The data may not be intrinsically important. But in the *context* of doing a situation assessment, the system may decide to integrate part of the un-

processed data before doing the situation assessment. Integrating the data first often adds value to the situation assessment because the system will have more information on which to base its assessment. This added value is added to the priority of the data integration plan request. The system goes back to the *prioritize problem* phase and if the data adds enough value to the situation assessment problem, it constructs an object to solve the integrate data problem. The data integration problem instance is added to the list of plan elements in the situation assessment plan object. The actual plan elements to integrate the data into the system are encapsulated within the data integration control plan object.

Heuristics used by the Control Planner

A real-time SA system is continuously supplied with sensor updates, pilot requests for information, anticipation of possible future events, and a plethora of cognitive actions that must be taken in order to assess the current situation. Thus, most of the work of the control part of the system is in deciding what problem to solve. Following are the five domain problems that the SA system solves:

1) Integrate new data into the domain blackboards
2) Focus sensors
3) Generate a current situation assessment
4) Generate a predicted assessment for some opportune time in the future
5) Generate a predicted assessment for time T seconds from the present

In order to solve the control problem, the SA system opportunistically chooses problems from among these five domain problems and makes efficient plans to solve them. This approach provides a built-in well-defined external interface to the system. Problems presented to the SA system by an external agent (e.g., an external system planner or the pilot) are placed in with the problems the SA system has presented to itself at the problem level of the control blackboard.

In order to illustrate the meta-level heuristics that the SA system uses to choose from among competing domain problems, we provide an example of how the SA system integrates new data into the domain blackboard (problem 1 above). The SA system places incoming sensor and intelligence data at the survey level of the assessment blackboard. This new data triggers a problem request at the problem level of the control blackboard. When the SA system decides to integrate the new data, the control rules make and carry out a plan that consists of an ordered sequence of domain modules which will be sequentially examined by the SA system.

All incoming information is rated for importance. EEOIs that have already been encountered by the system are given *Entity Assessment Ratings* (EARs):

$$\textbf{EAR} = [\textbf{Confirmed}(\text{danger}), \textbf{Plausible}(\text{danger})]$$

The EAR is a belief function that represents the degree to which an EEOI is a threat to the rotorcraft. The *confirmed* danger is the degree to which the system has confirmed that an EEOI is a danger to ownship. The *plausible* danger (or potential danger) is the worst-case danger that an EEOI presents to ownship at this time. Both of these numbers must lie in the range [0, 1]. The plausible danger is always greater than or equal to the confirmed danger. The less the system knows about an EEOI, the greater the difference between the plausible and the confirmed danger.

The EAR is synthesized into an *Interesting Rating*:

Interesting Rating = 0.7 * Confirmed danger + 1.3 * ability_garner() * (Plausible danger - Confirmed danger)

The system evaluates EEOIs as "more interesting" if there is a large gap between the plausible and confirmed dangers. This means EEOIs that might be dangerous but about which there is

little knowledge will be rated or ranked more interesting. Ability_garner is a function that calculates the degree to which the system thinks it currently has the ability to gather more information about the EEOI. When new data come in about an EEOI, we can use that entities' previously calculated Interesting Ratings to prioritize it.

When data about a previously unencountered entity arrives, SA the system favors the integration of the information into the domain blackboard if the data is about a nearby EEOI. The SA system especially favors it if automatic target recognition (ATR) has managed to already confirm an EEOI's vehicle type. Data about new EEOIs is considered intrinsically more important than data about previously encountered entities. The SA system attempts to reject duplicate information before any attempt is made to rate it or integrate it. The control planner always attempts to control the sensors to gain more information about interesting EEOIs.

The amount of time spent generating and evaluating heuristics must be balanced with the amount of time spent executing domain rules. It is possible to expend too many computational resources prioritizing problems and not enough time actively solving them. Entity Assessment Ratings and Interesting Ratings require processing resources for calculation. However, they must be calculated anyway for use by other parts of the system and these calculations are entirely procedural or algorithmic and are therefore computationally relatively inexpensive. Very little extra processing power is required to rate entities in this way. Such overlapping requirements often enable more sophisticated meta-level control knowledge to be produced. The results of the inferencing process represented at various blackboard levels by symbolic abstractions can thus be used as input for procedural/algorithmic computation that, in turn, produces useful metal-level control knowledge.

Using Dynamic Salience for Control

The planning approach to control has the disadvantage of always firing each of the rules that pertain to the chosen domain problem within the modules listed in the control plan. Another layer of control can be attained by directing the system to fire only the subset of eligible domain rules that best apply to the current domain problem. This flexibility is achieved within PalymSys™ by using an expanded form of the salience command.

The modifications to the salience command in PalymSys™ are based on the work done by Robert Orchard of the National Research Council of Canada in his extended version of CLIPS called BB_CLIPS [7]. The added syntax, called rule *features*, are descriptive characteristics of the knowledge contained in a rule that are placed within the antecedent of the rule.

```
;;
;; RULE: pred_pe2
;;
;;      If the EEOI will be able to see ownship and will be able to hit ownship and the EEOI's
;;       plan is combat_recon, main_attack, close_air_support, artillery, or guard then
;;       EE will probably be engaging you in the future (60%).  If not, then we can't be sure
;;       what the EEOI will do next (40%).
;;

(defrule pred_plan_element12
(declare                              ; feature list
        (salience 200)                ; salience type
        (reliability 35)              ; integer type
        (importance 25)               ; integer type
        (efficiency medium))          ; set type
(Module_focus (focus domain)(sub_focus pred_plan_element)(entity_focus all) (time_focus
?time&:(>= ?time3))(level policy)(BB CONTROL))
(object (is-a EEOI_Pred_location_long) (label ?name) (dist_from_ownship ?dist&:(< ?dist
6))(level interpretation)(BB PREDICTION)) ;; all EEs < 6km away.
(object (is-a EEOI_Pred_capability_long) (label ?name) (see_capability
?seecap&:(>= ?seecap .5)) (hit_capability ?hc&:(> ?hc .5))(level pred_cap)(BB ASSESSMENT))
(object (is-a EEOI_Plan) (label ?name) (propagation ?prop)
```

```
        (type ?type&:(member$ ?type (create$ combat_recon long_range_recon guard_forward
guard_flank guard_rear main_attack close_air_support)))(level plan_interp)(BB ASSESSMENT))
=>
        (assert_belief ?name pred_plan_element ?prop ".6 ENGAGE_OWNSHIP .4 ALL")
)
```

Each feature has an associated dynamic salience value determined by its feature arguments. PalymSys™ has a combining function that evaluates the salience of each rule between rule firings. The feature argument itself is a pointer to a dynamic data structure of salience values that is modified by control rules at run-time. For instance, if the system is suddenly faced with a time constraint, a control decision can be made to globally raise the value of the efficiency feature.

A new feature in CLIPS 6.0 is the (set-salience-evaluation every-cycle) command which enables salience values to be calculated dynamically at run-time between rule firings. It is possible to achieve the same functionality described above from within the CLIPS 6.0 shell by placing a function as the argument for the salience command. Between rule firings, the function dynamically computes salience values which are based on global control variables whose values have been determined by control decisions.

A Hybrid PalymSys™/C++ Belief Network

Reasoning under uncertainty is a necessity for a situation assessment system. The SA system must make prescient inferences about such things as an EEOI's plan or the associated elements (steps) in that plan. The EEOI's plans and plan elements cannot be known for certain until various activities are explicitly observed. Other inferences, such as an EEOI's intent or an EEOI's *predicted* plan element, can never be known for certain. Hypotheses must be based on incomplete and unreliable evidence because the battlefield is a complex, uncertain environment. Reasoning under uncertainty requires a probabilistic model of reasoning that supports reasoning using contradictory and partially contradictory hypotheses in which the system has varying degrees of confidence.

PalymSys™ enables the user to construct a Dempster-Shafer (D-S) belief network quite easily. A command line interface allows the user to specify size, structure and number of instances of the network at run time. A belief network propagates the uncertainty associated with a particular piece of knowledge throughout the entire hierarchy of hypotheses that depend upon it. The SA control planner in conjunction with the Rete Pattern Matching algorithm handles the belief propagation through the system hierarchy. When rules are added to the system, no modifications of existing C++ or PalymSys™ code are necessary. By placing a single function call on the consequent of the added rule(s), the system will incorporate the new rule(s) into the belief network automatically. A formal explanation of Dempster-Shafer theory is beyond the scope of this paper. Such detailed presentations can be found in [8, 9, 10]. However, the CLIPS modifications described here can be applied to a monotonic, feed-forward belief network of any type (i.e., Bayesian).

A frame of discernment is a set of mutually exclusive hypotheses. Exactly one hypothesis in a frame of discernment is true at any one time. Each module that uses D-S reasoning in SA has its own frame of discernment. For example, the frame of discernment corresponding to the Plan Element module is the set of fourteen distinct plan elements that an entity is capable of performing within the context of all possible plans. When an entity is encountered, it is assumed to be performing one and exactly one of these plan elements. The purpose of the plan element module is to assign belief values to each of the members of the plan element frame of discernment. A set of propagation values for the plan element frame of discernment is also calculated. The propagation values serve as input to other frames of discernment that use plan elements as evidence.

Recall that the SA system follows entity specific control plans in order to integrate new data into the system. A control plan is an ordered list of modules that the system will sequentially visit to solve a domain problem. The exact order of the control plan will vary depending on what type of data is being integrated into the system. When a control plan element is executed, a particular

module fires its rules, assigns belief to the members of its frame of discernment, and then control proceeds to the next module in the control plan which is typically on the next higher level within the domain blackboard hierarchy.

A typical SA domain rule within the belief network will look much like the following rule from the Plan module:

```
RULE: plan_rule12
Description:
;; If the EE's current Plan Element is Reporting then his Plan might be (40%)
;; surveillance, combat recon, or long range recon.  It also might be, to a
;; slightly lesser degree (30%), guard forward, guard flank, or guard rear.  If
;; it's not in those two sets, then the EE could be performing any Plan (30%).
(defrule plan_rule12
(Module_focus (focus domain)(sub_focus plan)(entity_focus ?name)
       (level policy)(BB CONTROL))
(object (is-a EEOI_Plan_element) (label ?name) (type report)
       (propagation ?prop_value&:(> ?prop 0))(level plan_interp)(BB ASSESSMENT))
=>
(assert_belief ?name Plan_Module ?prop_value ".4 SUR CRP LRRP .3 GFR GFL GR .3 ALL")
)
```

The variable *name* is needed as a tag because SA makes an instance of the belief network for each new EEOI encountered. In this case, the assert_belief function places the belief into the frame of discernment associated with the Plan module with a propagation value of *prop_value*. The Module_focus template values in the rule will allow this rule to fire only when the system is firing the rules on the assessment blackboard in the plan module. In this way, the control plan assures that the rules in each frame of discernment for a particular EEOI are finished firing before advancing up to higher levels on the blackboard. The same technique can be used with any monotonic feed-forward blackboard belief propagation scheme.

When evidence is obtained from another frame of discernment, as is the case with the EEOI_plan_element object in the rule above, the propagation value is also sent to the assert_belief function. Evidence without a propagation value associated with it is assumed to have a probability of truth of one. However, the system can easily adapt to any uncertain data by attaching a propagation value to them.

The last rule fired within each module calls the function get_belief to access the appropriate belief and propagation values for the module's frame of discernment. The propagation values will be passed to the next level up in the blackboard hierarchy via the propagation slot in the object that corresponds to the current entity and the current blackboard level.

New domain rules are easily added to the system by placing the assert_belief function on their RHS. No modifications to the reasoning under uncertainty function code are required since propagation is handled entirely by the Rete Pattern Matching algorithm and the SA control planner.

Simulation and Test Environment Interface

SA uses interprocess communication to communicate with our rotorcraft mission simulation and test environment (STE). The STE is a graphical simulation test bed written in C++ and implemented on a RS/6000 workstation. During simulation runs, the STE sends SA sensor information and SA sends the STE directional parameters to control the sensors. A communications class was written for the STE and integrated with PalymSys™. Shared memory in our system is accessed via the standard system C libraries<*sys/shm.h*> and <*sys/types.h*>. More specifically, the STE uses the function calls *shmat*, *shmget*, *shmdt* to attach a process, grab the shared memory and detach the process, respectively.

Standard CLIPS terminates when the agenda is empty. PalymSys™ can be directed to run continuously even though the agenda is empty by adding an optional argument to the run command. The inference engine will idle, waiting for facts to be asserted into the system. This capa-

bility is essential whenever the system depends on an independent process, like the STE, as a source for fact assertions.

Three functions were embedded into PalymSys™ in order to communicate with the STE. One function checks the communications link to see if information had been passed over from the STE. The maximum buffer size was 200 characters, so the PalymSys™ function got the name of a file from shared memory that had just been created by the STE. Another PalymSys™ function reads the file just created by the STE and asserts the contents as facts into the PalymSys™ fact base using the *AssertString* CLIPS C library call. Finally, another function lets the STE know via shared memory when SA has sent it information via file transfer. This two-way real-time interprocess communication provides a realistic simulation of a rotorcraft environment.

Summary and Conclusions:

We have implemented a situation assessment blackboard expert system in PalymSys™ -- an extended version of CLIPS. Blackboards are an excellent paradigm for CLIPS expert system implementations. The control blackboard architecture is especially well-suited to real-time applications like SA. We developed a control planner in PalymSys™ that chooses the most important problems to solve based on complex meta-level situation characteristics. The control planner creates domain plans to solve the problems that it chooses. SA uses a monotonic feed-forward Dempster-Shafer belief network implemented in C++. The size and number of instances of the network is dynamic and completely controlled at run-time from the PalymSys™ shell. Finally, we interfaced the SA system to our Simulation and Test Environment using interprocess communication techniques. A continuous run feature was added which enables the inference engine to idle even when the agenda is empty.

Acknowledgments

Mr. Joe Marcelino was instrumental in the implementation of the assessment and prediction blackboards. Mr. Richard Warren implemented the terrain reasoning system. Mr. Jerry Clark provided invaluable advice on the reasoning under uncertainty mechanism used in SA. Mr. Clark served as the rotorcraft domain expert on this project. Mr. Steve Schnetzler implemented the interprocess communications between SA and the Simulation and Test Environment.

Bibliography

[1] D. Ballard and L. Rippy, "A knowledge-based decision aid for enhanced situational awareness," in *Proceedings of Thirteenth Annual Digital Avionics Systems Conference*, Phoenix, AZ, 1994, in press.

[2] H. P. Nii, "Blackboard Systems - Part I," *AI Magazine*, vol. 7, no. pp. 38 - 53, 1986.

[3] H. P. Nii, "Blackboard Systems - Part II," *AI Magazine*, vol. 7, no. 4, pp. 82 - 107, 1986.

[4] N. Carver and V. Lesser, "A planner for the control of problem-solving systems," *IEEE Transactions on Systems, Man and Cybernetics*, vol. 23, no. 6, pp. 1519 - 1536, 1993.

[5] B. Hayes-Roth, "Opportunistic control of action in intelligent agents," *IEEE Transactions on Systems, Man and Cybernetics*, vol. 23, no. 6, pp. 1575 - 1587, 1993.

[6] B. Hayes-Roth, "A blackboard architecture for control," *Artificial Intelligence*, vol. 26, pp. 251 - 321, 1985.

[7] R. Orchard and A. Diaz, "BB_CLIPS: Blackboard extensions to CLIPS," in *Proceedings of First CLIPS Conference*, Houston, Texas, 1990, pp. 581 - 591.

[8] J. Pearl, Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference. San Mateo, CA: Morgan Kaufmann, 1988.

[9] G. Shafer, A Mathematical Theory of Evidence. Princeton, NJ: Princeton University Press, 1976.

[10] G. Shafer and J. Pearl, Uncertain Reasoning. San Mateo, CA: Morgan Kaufmann, 1990.

S21-61
34104
P-13

N95-19768

DYNACLIPS (DYNAmic CLIPS): A DYNAMIC KNOWLEDGE EXCHANGE TOOL FOR INTELLIGENT AGENTS

Authors

Yilmaz Cengeloglu
PO Box 4142, Winter Park, FL, 32793-4142
E-mail: yil@engr.ucf.edu

Soheil Khajenoori, Assoc. Prof., Embry-Riddle Aeronautical University,
Department of Computer Science,
Daytona Beach, FL, 32114, E-mail: soheil@erau.db.erau.edu

Darrell Linton, Assoc. Prof., University of Central Florida,
Department of Electrical and Computer Engineering,
Orlando, FL, 32816, E-mail: dgl@engr.ucf.edu

ABSTRACT

In a dynamic environment, intelligent agents must be responsive to unanticipated conditions. When such conditions occur, an intelligent agent may have to stop a previously planned and scheduled course of actions and replan, reschedule, start new activities and initiate a new problem solving process to successfully respond to the new conditions. Problems occur when an intelligent agent does not have enough knowledge to properly respond to the new situation. DYNACLIPS is an implementation of a framework for dynamic knowledge exchange among intelligent agents. Each intelligent agent is a CLIPS shell and runs a separate process under SunOS operating system. Intelligent agents can exchange facts, rules, and CLIPS commands at run time. Knowledge exchange among intelligent agents at run time does not effect execution of either sender and receiver intelligent agent. Intelligent agents can keep the knowledge temporarily or permanently. In other words, knowledge exchange among intelligent agents would allow for a form of learning to be accomplished.

1. INTRODUCTION

Applications of expert systems to variety of problems are growing rapidly. As the size and complexity of these systems grow, integration of independent cooperating expert systems is becoming a potential solution approach to large scale applications. In this paper, the blackboard model of distributed problem solving is discussed and architecture, implementation and usage of DYNACLIPS is explained.

1.1. Distributed Problem Solving (DPS)

Distributed problem solving in artificial intelligence is a research area which deals with solving a problem in a distributed environment through planning and cooperation among a set of intelligent entities *(i.e., agents)*. Each intelligent agent can run in parallel with other intelligent agents. Intelligent agents may be geographically distributed or operate within a single computer. An intelligent agent may possess simple processing elements or a complex rational behavior. A paramount issue in DPS is the communication and information sharing among participating intelligent agents, necessary to produce a solution. The blackboard model of problem solving is one of the most common approaches in the distributed artificial intelligence area. In the following section we will focus on blackboard architecture as a model for distributed problem solving.

1.2. Blackboard Model of Distributed Problem Solving

The blackboard architecture (BBA) is one of the most inspiring cooperative problem solving paradigms in artificial intelligence. The approach had generated much interest among researchers. BBA is a relatively complex problem solving model prescribing the organization of knowledge, data and the problem-solving behavior within an overall organization. The blackboard model is becoming a useful tool for complex applications whose solution requires a set of separate though interrelated sources of knowledge and expertise.

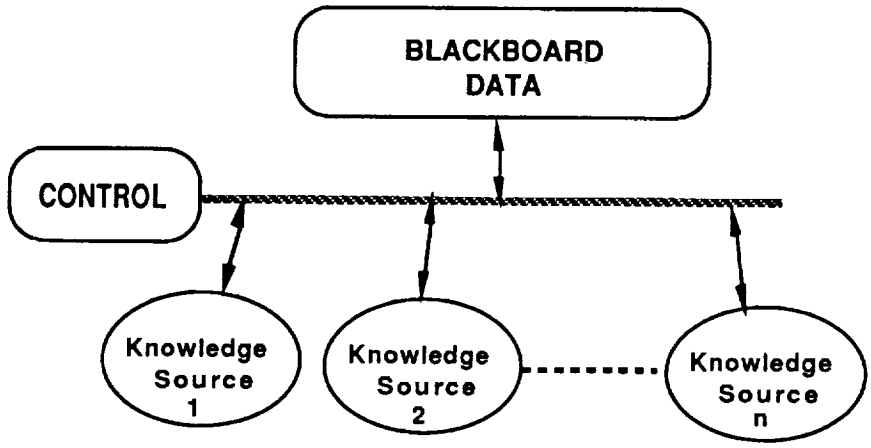


Figure 1. Blackboard Model For DSP

A blackboard model contains blackboard, control, and knowledge sources. Figure 1 shows a basic blackboard model for DPS applications. The Knowledge Sources are the knowledge needed to solve the problem; they are kept separate and independent. Each knowledge source can use different knowledge representations techniques. The Blackboard Data is a global database that contains problem-solving states. The knowledge sources produce changes to the blackboard that lead incrementally to the solution of the problem being solved. Communication and interaction among the knowledge sources takes place solely through the blackboard. The Control determines the area of the problem solving space on which to focus. The focus of attention can be either the knowledge sources, the blackboard or a combination of both. The solution is built one step at a time by using a cooperation of different knowledge sources. The blackboard model is a complete parallel and distributed computation model. The parallel blackboard model involves the parallel execution of knowledge sources and the control component. The distributed blackboard model involves the communication of blackboard data among blackboard subsystems. The main issue here is to decide what to communicate and where and when to send data. The blackboard systems are being used increasingly in real time systems. Architectural extensions to the basic blackboard model can be added to increase its performance for real time applications. [1,2,3]

1.3. An Overview of Existing BBA Tools

A number of BBA tools are reported in the literature. Here we provide an overview of some of these systems.

BB_CLIPS (Blackboard CLIPS) [4, 5] is an extended version of CLIPS version 4.3 developed by National Research Council of Canada. In BB_CLIPS, each CLIPS rule or group of rules serves as a knowledge source. The fact base of BB_CLIPS serves as the blackboard, and its agenda manager serves as the scheduler.

RT-1 architecture [6] is a small-scale, coarse-grained, distributed architecture based on the blackboard model. It consists of a set of reasoning modules which share a common blackboard data and communicate with each other by "signaling events".

PRAIS (Parallel Real-Time Artificial Intelligence Systems) [7] is an architecture for real-time artificial intelligence system. It provides coarse-grained parallel execution based upon a virtual global memory. PRAIS has operating system extensions for fact handling and message passing among multiple copies of CLIPS.

MARBLE (Multiple Accessed Rete blackboard linked Experts) [8] is a system that provides parallel environment for cooperating expert systems. Blackboard contains facts related to the problem being solved and it is used for communication among expert systems. Each expert shell in the system keeps a copy of the blackboard in its own fact base. Marble has been used to implement a multi-person blackjack simulation.

AI Bus [9] is a software architecture and toolkit that supports the construction of large-scale cooperating systems. An agent is the fundamental entity in the AI bus and communicates with other agents via message passing. An agent has goals, plans, abilities and needs that other agents used for cooperation.

GBB (Generic Blackboard) [10,11] is toolkit for developers needs to construct a high-performance blackboard based applications. The focus in GBB is increasing the efficiency of blackboard access, especially for pattern-based retrieval. GBB consist of different subsytems : a blackboard database development subsystem, control shells, knowledge source representation languages and graphic displays for monitoring and examining blackboard and control components. GBB is an extension of Common Lisp and CLOS (Common Lisp Object System).

GEST (Generic Expert System Tool) [12] has been developed by Georgia Tech Research Institute. The main components of the GEST are the central blackboard data structure, independent experts or knowledge sources and the control module. The blackboard data structure holds the current state of the problem solving process. It is also common communication pathway among knowledge sources.

CAGE and POLIGON [13] have been developed at Stanford University. They are two different frameworks for concurrent problem solving. CAGE is a conventional blackboard system which supports parallelism at knowledge sources level. The knowledge source, the rules in the knowledge source, or clauses in a rule can be executed in parallel. CAGE is a shared memory multiprocessing system. POLIGONs' functionality is similar to CAGE.

Hearsay-II [2,3,14] is a speech understanding system developed at Carnegie-Mellon University. Hearsay-II provides a framework that different knowledge sources cooperate to solve a problem.

More recently, significant work is being done to develop Knowledge Interchange Formats (KIF) and Knowledge Query and Manipulation languages (KQML) by Stanford University [15,16]. KIF is a computer-oriented language for the interchange of knowledge among disparate programs that is written by different programmers, at different times, in different languages. KIF is not a language for the internal representation of knowledge. When a program reads a knowledge base in KIF, it converts the knowledge into its own internal form. When the program needs to communicate with another program, it maps its internal data structures into KIF. KQML messages are similar to KIF expressions. Each Messages in KQML is one piece of a dialogue between the sender and receiver programs.

2. IMPLEMENTATION OF DYNACLIPS

Using the SunOS operating system multiprocessing techniques and interprocess communication facilities, we have developed a prototype system on a Sun platform to demonstrate the dynamic

knowledge exchange among intelligent agents. In the following sections we provide a short overview of SunOS InterProcess Communication Facilities (IPC) and describe the implementation aspect of the DYNACLIPS.

2.1. InterProcess Communication Facilities (IPC)

Interprocess Communication involves sharing data between processes and coordinating access to shared data. The SunOS operating system provides several facilities and mechanism by which processes can communicate. These include Messages, Semaphores and Shared Memory.

The *Messaging facility* provides processes with a means to send and receive messages, and to queue messages for processing in an arbitrary order. Messages can be assigned specific types and each would have an explicit length. Among other uses, this allows a server process to direct message traffic between multiple clients on its queue. Messages sent to the queue can be of variable size. The application programmer must insure that the queue space limitations are not exhausted when more than one process uses the same queue. The process owning the queue must establish the read/write permissions to allow/deny other processes access to the queue. Furthermore, it is the responsibility of the owner process to remove the queue when it is no longer in use or prior to exiting.

Semaphores provide a mechanism by which processes can query or alter status information. They are often used to monitor and control the availability of system resources, such as shared memory segments. Semaphores may be operated as individual units or as elements in a set. A semaphore set consists of a control structure and array of individual semaphores.

Shared memory allows more than one process at a time to attach a segment of physical memory to its virtual address space. When write access is allowed for more than one process, an outside protocol or mechanism such as a semaphore can be used to prevent contentions [17].

2.2. Architecture of DYNACLIPS

In this section we provide an overview and discussion of each component of the DYNACLIPS. Figure 2 represents the overall architecture of the DYNACLIPS. Shared memory has been used to implement the system main blackboard to broadcast messages from control to intelligent agents. Message queues have been used to transfer messages from the intelligent agents to the control. Semaphores are used to make intelligent agents and control to sleep or wake up in order to reduce the load on CPU. Each intelligent agent and control has an input/output port to the world outside of the application framework to interface with the other processes outside of their environment. These outside processes might be any program that uses the application framework. In DYNACLIPS, intelligent agents and control can also use IPC facilities to interface with the outside programs.

2.3. Control

The control component of the system has been implemented using the C programming language. It runs as a separate process and communicates with the other processes using IPC facilities. The control can be loaded and executed by entering the word "control" at SunOS prompt. The control always has to be loaded first. Once the control is loaded, it creates the three incoming control message queues for the requests coming from the intelligent agents, one shared memory to be used as the main system blackboard and two semophores for control and intelligent agents. The control has read/write access to the main system blackboard and has only read access from the message queues. If there is no request from intelligent agents, control sleeps until any agent makes a request.

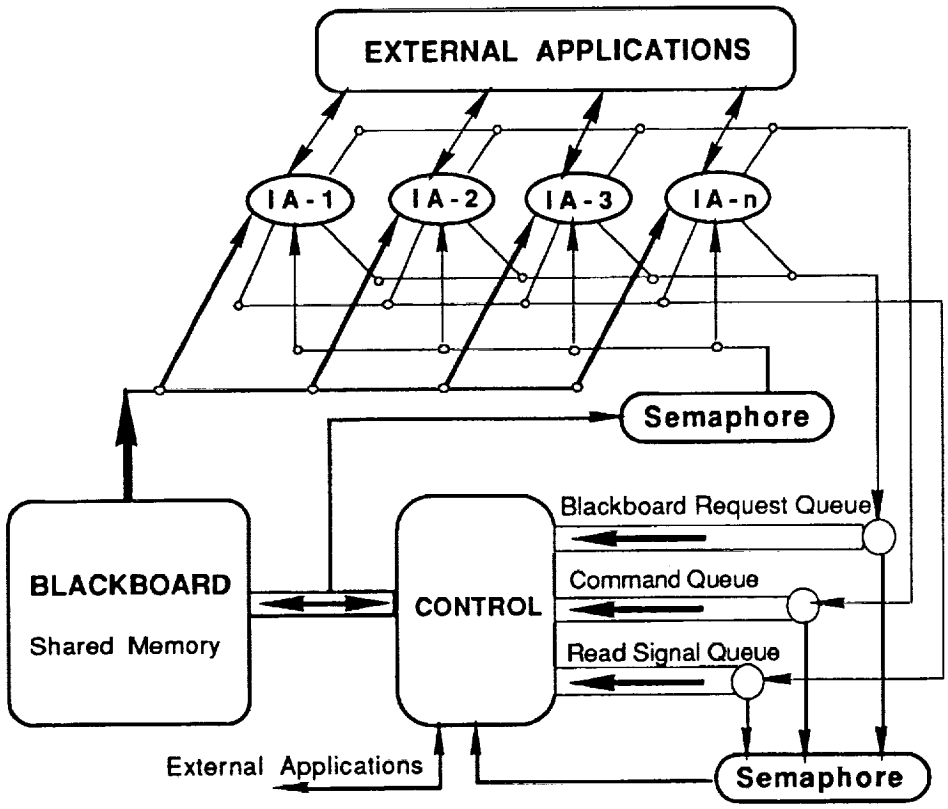


Figure 2. Architecture of DYNACLIPS

The DYNACLIPS takes advantage of the multiprocessing capabilities of the SunOS operating system. A message facility has been used for setting up a communication link from the intelligent agents to the control. The control creates incoming queues to receive messages from the intelligent agents. These control message queues use "First-In First-Out" (FIFO) methodology as shown in Figure 3.

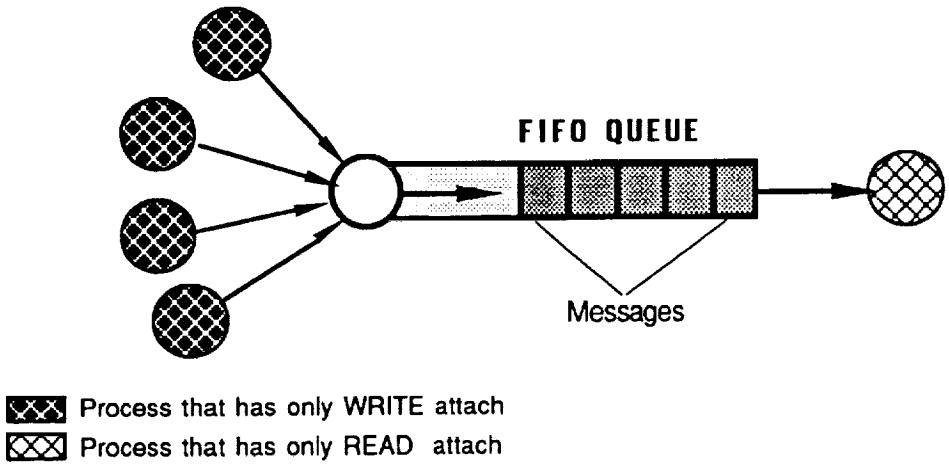


Figure 3. Message Transfer Among Processes

In order to communicate with the control process, intelligent agents send their requests to the control message queues. Intelligent agents have only write permission to these queues. Message queue facility of IPC can also be utilized to allow the control module and/or intelligent agents to communicate with other application programs running outside of their environment.

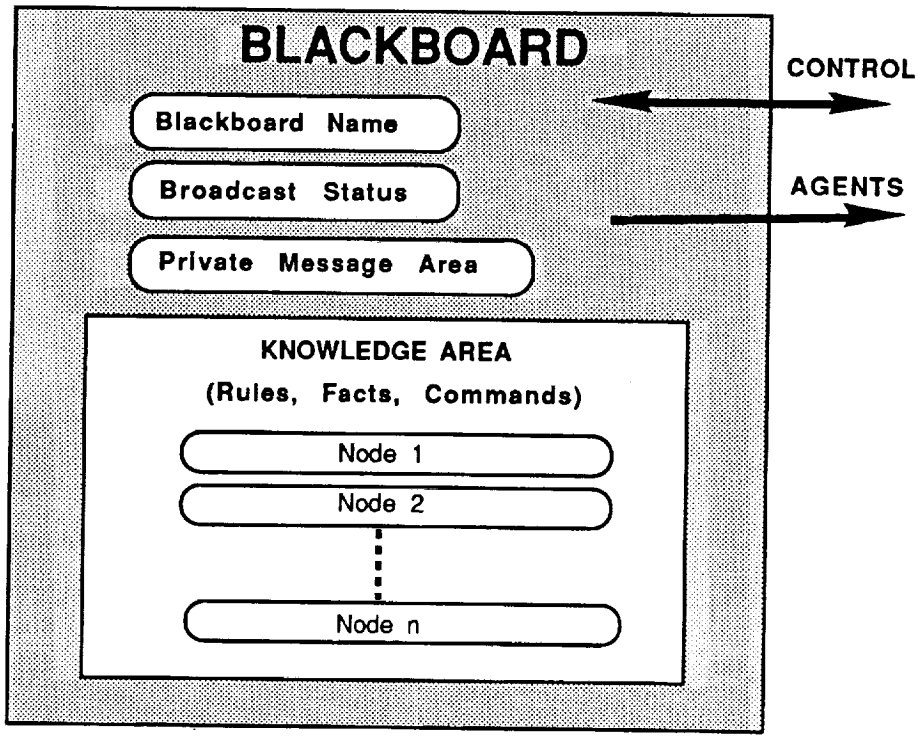


Figure 4. Blackboard

2.4. Blackboard

The blackboard is a shared memory that holds information and knowledge that intelligent agents use. The blackboard is organized by the control component. The blackboard has four different members as shown in Figure 4. These members are: 1) *Blackboard name* which holds the name of the blackboard that is being used, this is necessary if multiple blackboards are in use, 2) *Blackboard status* which is a counter that is incremented after each update of the blackboard, 3) *Private message area* which is used by the control component to send system related messages to the intelligent agents, 4) The *knowledge area* which holds, rules, facts and commands that needs to be broadcast to the intelligent agents.

Shared Memory facility of IPC has been used to broadcast messages to all intelligent agents via the control. Figure 5 represents broadcasting data using shared memory configuration. When a process that has a write attach, writes a message to the shared memory, this message will be visible to all processes that have read attach to shared memory. When there is more than one process able to write to shared memory, semaphores should be used to prevent processes accessing the same message space at the same time. In DYNACLIPS semaphores were not used for shared memory, because the control component is the only process that has read and write accesses to the shared memory and intelligent agents can only read from the shared memory.

In DYNACLIPS, there is a local blackboard for each intelligent agent and one main blackboard for the control. The DYNACLIPS uses the shared memory facility to implement the main system blackboard. This implementation was possible because all intelligent agents run on the same computer. If intelligent

agents were distributed on different computer systems, then a copy of the main blackboard needs to be created and maintained in each computer system.

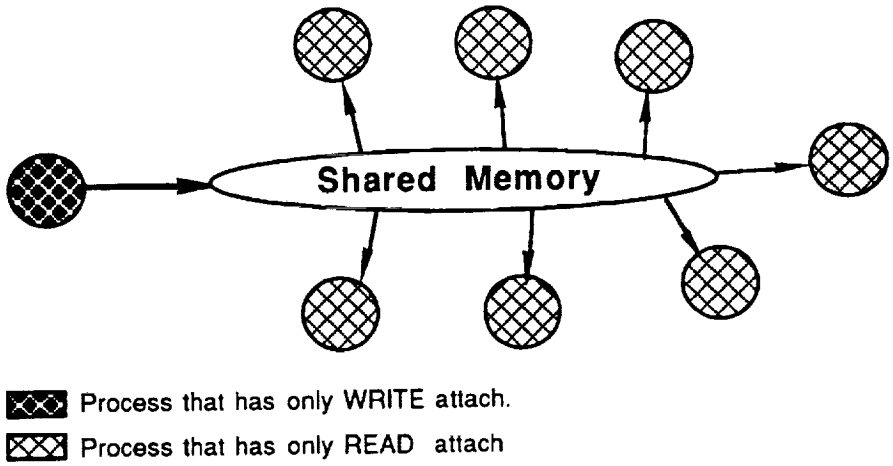


Figure 5. Message Broadcasting With Shared Memory

2.5. Intelligent Agents

An intelligent agent can be any application, such as an expert system shell, that is able to use IPC facilities on the SunOS operating system. In this prototype system, CLIPS (C Language Production System) was used as intelligent agents. CLIPS [18] is an expert system shell which uses a rule-based knowledge representation scheme. We have extended the CLIPS shell by adding a set of functions to provide the necessary capabilities to use IPC facilities.

The following command should be executed at SunOS prompt to load and activated a CLIPS based intelligent agent.

dynaclips <Knowledge base name > <Intelligent agent name> [Group name]

The *Knowledge base name* is a file that contains CLIPS rules. The *Intelligent agent name* and *Group name* are names given to intelligent agent by the user or application program that loads this knowledge base. Multiple copies of the CLIPS expert system shell, each representing an agent, can be loaded and activated at the same time using the above command. *Intelligent agent name* and *Group name* are also inserted as initial fact to intelligent agents. Following will be initial facts in the intelligent agents:

(knowledge name is *<Intelligent agent name>*)
(member of group *<Group name>*)

Once an intelligent agent is loaded, 1) it finds out the names of the control message queues and attaches itself to the queues as a writer. 2) It finds out the name of the main blackboard and attaches itself to it as a reader. 3) It sends a message to control component to inform that it has joined the system. If there is no rule to fire and nothing is changed in the blackboard, an intelligent agent sleeps until something changes in the blackboard.

As we mentioned previously, we have added number of functions to the CLIPS shell. The following functions can be used by an intelligent agent to communicate with the control as well as with other intelligent agents.

(ADD_BB_STR *<message string>)*
(DEL_BB_STR *<message string>)*
(CHANGE_GROUP_NAME *<new_group_name>)*
(EXIT_CLIPS)

ADD_BB_STR *(Add BlackBoard String)* is called when an intelligent agent wants to add a message to the main blackboard. DEL_BB_STR *(Delete BlackBoard String)* is called when an intelligent agent wants to delete a message from the main blackboard. CHANGE_GROUP_NAME is called when intelligent agent needs to change group. EXIT_CLIPS is called when an intelligent agent is exiting from the system permanently. In the above commands, *Message string* takes the following format :

"*<destination> <type> <message>*"

The *Destination* field should be the name of an intelligent agent or group currently active in the system, or "ALL" which specifies that the message should be received by all active intelligent agents. *(In the above format, blank characters were used to separate each field.)*

The *Type* field should be "FACT", "RULE" or "COMMAND", which describes the type of the message in the *message string*. If the *type* is FACT, the *message* will be added or deleted, depending on the functions, to shared fact base of the intelligent agent specified by the *destination* field. If the *type* is RULE, the *message* will be added or deleted to the dynamic knowledge base of the intelligent agent(s) as specified in the *destination* field. If the *type* is COMMAND, then *message* will be executed as a command by the intelligent agent specified in the *destination* field. Commands are always removed from the main blackboard after intelligent agents receive a copy of the blackboard.

Message can contain facts, rules or commands. Since the current implementation of the prototype system only uses CLIPS shell to represent an intelligent agent, we have chosen to follow the syntax of the CLIPS to represent facts, rules or commands. If another expert system shell is used to represent an intelligent agent, this common syntax should be observed to transfer facts, rules or commands. It is the intelligent agent's responsibility to translate this common syntax to its own internal syntax.

The following sections describe, in more detail, the process of transferring facts, rules, and commands among intelligent agents.

2.6. Fact Transfer Among Intelligent Agents

An intelligent agent can send fact(s) to an individual intelligent agent, group of intelligent agents or to all intelligent agents in the system. Facts stay in the main blackboard until removed by the sender intelligent agent or by other intelligent agent(s) that has/have the permission to delete the fact. ADD_BB_STR and DEL_BB_STR commands are used to insert, or remove fact(s) from the main blackboard. The following examples show how facts transferred among intelligent agents in the system.

Given the following information :

?y is a string containing "ALL FACT Hi there, I have just joined the system"
?x is a string containing "IA-2 FACT apple is red"
IA-2 is the name of an intelligent agent active in the system.

Then:

(ADD_BB_STR ?y) adds the message *(i.e., the fact) "Hi there, I have just joined the system"* to the main blackboard as a fact and all intelligent agents insert this fact to their internal shared fact base.

(DEL_BB_STR ?y) deletes the message *(i.e., the fact) "Hi there, I have just joined the system"* from the main blackboard as well as from internal shared fact base of any intelligent agent that has this fact in its shared fact base.

(ADD_BB_STR ?x) adds *"apple is red"* to the main blackboard as a fact and only IA-2 is allowed to insert this fact to its shared fact base.

(DEL_BB_STR ?x) deletes *"apple is red"* from the main blackboard and causes IA-2 to remove this fact from its shared fact base.

2.7. Command Transfer Among Intelligent Agents

An intelligent agent can send command(s) to an individual intelligent agent, group of intelligent agents or to all intelligent agents in the system. Commands do not remain on the main blackboard, the receiver intelligent agent executes the command immediately upon its arrival. Commands are deleted by the receiver intelligent agent(s) as soon as they are executed. ADD_BB_STR function should be used for transferring commands among intelligent agents. DEL_BB_STR function is not available on COMMAND type. Following examples demonstrate how commands are transferred among intelligent agents. In the following examples all the commands are standard commands available in the CLIPS expert system shell.

Given the following information :

?y is a string containing "ALL COMMAND (rules)"
?x is a string containing "IA-2 COMMAND (watch facts)"
?z is a string containing "GROUP1 COMMAND (CHANGE_GROUP_NAME "GROUP2")"
IA-2 is the name of the an intelligent agent active in the system.

Then:

(ADD_BB_STR ?y) all intelligent agents in the system execute *(rules)* command which means print all rules in the intelligent agent knowledge base.

(ADD_BB_STR ?x) IA-2 executes *(watch facts)* command.

(ADD_BB_STR ?z) all intelligent agents in the GROUP1 will change their group name to GROUP2.

All CLIPS commands are supported by the DYNACLIPS. Hence, an intelligent agent can modify the knowledge of other intelligent agents via sending the appropriate command. Application programmer should be careful when designing the system since it is possible to remove static knowledge and local facts of the intelligent agent receiving the commands.

2.8. Rule Transfer Among Intelligent Agent

An intelligent agent can send rule(s) to an individual intelligent agent, group of intelligent agents or to all intelligent agents in the system. Rules stay on the main blackboard until removed by the sender intelligent agent or other intelligent agent(s) that has the right permission to delete the rule. CLIPS format should be followed to represent rules. The type RULE should be used in the type field of the message string. ADD_BB_STR and DEL_BB_STR commands can be used to add or delete rules from the main blackboard. The following presents examples for transferring rule among intelligent agents.

Given the following information :

?x is a string containing "IA-2 RULE *(defrule* ***rule1*** *(apple is red) => (facts))* "
?y is a string containing "ALL RULE *(defrule* ***rule2*** *(red is color) => (facts))* "

IA-2 is the name of an intelligent agent active in the system.

Then :

(ADD_BB_STR ?x) rule1 will be added to the main blackboard and only IA-2 can insert the *rule1* into its dynamic knowledge base.

(ADD_BB_STR ?y) rule2 will be added to the main blackboard and all intelligent agents can insert *rule2* into their dynamic knowledge base.

2.9. Knowledge Transfer Among Intelligent Agents

Knowledge can be exchanged among intelligent agents by using combination of facts, rules and commands transfers. Different methodologies can be used for knowledge transfer; knowledge can be exchanged among intelligent agents in temporary or permanent bases.

Under temporary knowledge transfer option, the sender intelligent agent specifies the rule(s) that needs to be transferred as well as specifying when the rules needs to be removed from dynamic knowledge base of the receiver intelligent agent. The following example shows how to transfer a temporary knowledge among intelligent agents.

Given the following information :

?x is a string that contains the following :

" ALL COMMAND (defrule rule1 (lights are on) =>

(turn of the lights)

(assert (lights are off))) "

?y is a string that contains the following :

" ALL COMMAND (defrule rule2 (lights are off) =>

(undefrule rule1)

(undefrule rule2)) "

Then:

(ADD_BB_STR ?x)

(ADD_BB_STR ?y)

In the above example, two rules were broadcasted to all intelligent agents in the system. All intelligent agents will insert these two rules into their dynamic knowledge base. The rules will remain in the intelligent agent's dynamic knowledge base until *rule1* is fired. *Rule2* will be fired after *rule1*, which would cause *rule1* and itself to be removed from the intelligent agents' dynamic knowledge bases. Using type COMMAND will cause that the two rules be deleted from the main blackboard as soon as all intelligent agents have read them into their dynamic knowledge base. By eliminating the second function *(i.e., (ADD_BB_STR ?y))* from the previous example, the rule can be placed permanently in the receiver intelligent agents' knowledge bases. Hence, the knowledge encoded in the rule can be used by the receiver intelligent agent from that point on. The following example demonstrates this concept.

Given the following information

?x is a string that contains the following :

" ALL COMMAND (defrule rule1 (lights are on) =>

(turn of the lights)

(assert (lights are off)) "

Then:

(ADD_BB_STR ?x)

Type RULE should be used to transfer knowledge when the intelligent agents are joining, exiting or re-joining the framework continuously. In this case, knowledge stays in the main blackboard until deleted explicitly by the sender intelligent agent.

3. CONCLUSIONS AND FURTHER STUDIES

By introducing simple communication protocols among intelligent agents, we have introduced a framework through which intelligent agents can exchange knowledge in a dynamic environment. Using the DYNACLIPS common knowledge can be maintained by one intelligent agent and broadcasted to the other intelligent agents when necessary.

In a dynamic environment, intelligent agents must be responsive to unanticipated conditions. When such conditions occur, an intelligent agent may be required to terminate previously planned and scheduled courses of action, and replan, reschedule, start new activities, and initiate a new problem solving process, in order to successfully respond to the new conditions. Problems occur when an intelligent agent does not have sufficient knowledge to properly respond to the new condition. In order to successfully respond to unexpected events in dynamic environments, it is imperative to have the capability of dynamic knowledge exchange among intelligent agents.

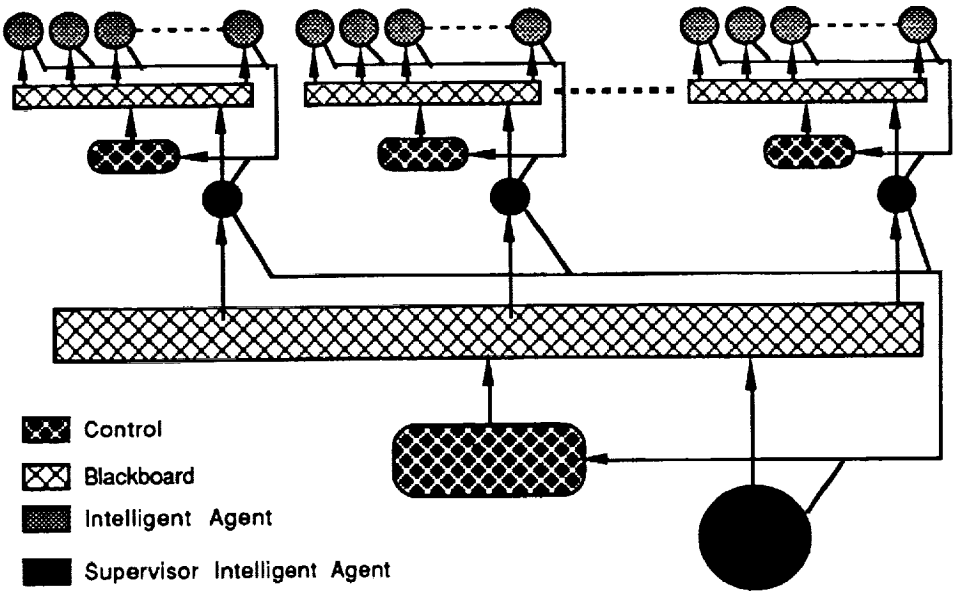


Figure 6. An Example of a Framework which Includes Multilayer Aspects

We believe that dynamic knowledge exchange would be an important feature for any application in which unanticipated conditions or events occur. Using the proposed dynamic knowledge exchange capability, cooperative problem solving sessions can be initiated where each intelligent agent can share its problem relevant knowledge with other intelligent agents to resolve the problem. An obvious advantage

of this capability is the elimination of redundant knowledge and hence the improved utilization of the system memory capacity. In addition, by using this framework a form of learning can take place and thus additional problem solving knowledge is created.

The basic framework presented in this research could be extended to include a multilayer environment. This can be done by providing supervisory blackboard and control components to create a single entity by combining separate frameworks. (See Figure 6.)

The proposed framework can easily be expanded to accommodate more than one blackboard when it is necessary. In this case one control module is associated with each blackboard as in Figure 7. The basic process of communication with the control modules would be the same as presented in the previous sections.

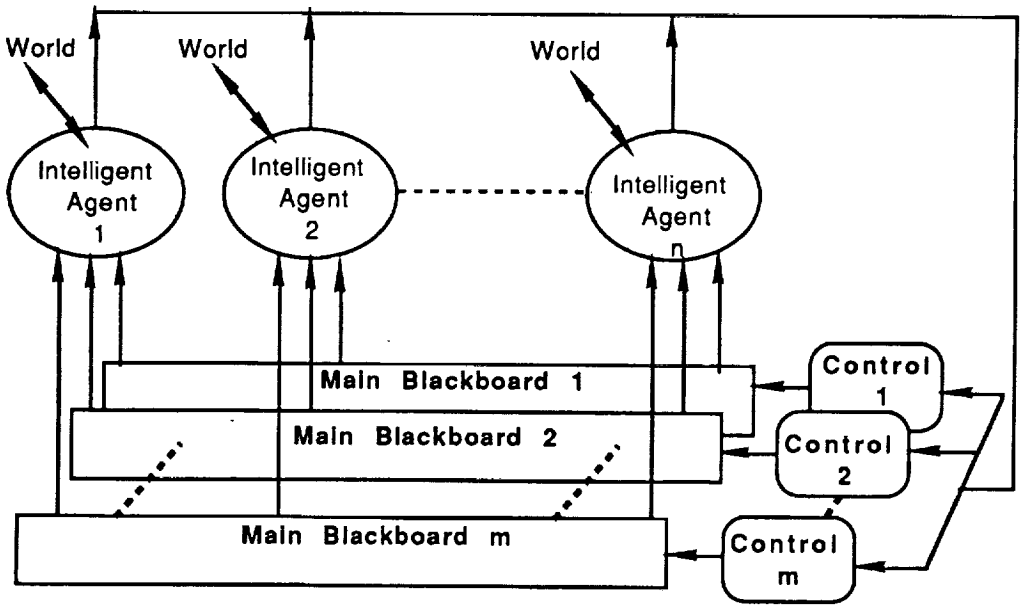


Figure 7. Using Multiple Blackboard

The prototype system is currently running on a single computer system using multiprocessing facilities. By using networking, it is possible to extend the system functionality to support distributed environments. Hence, intelligent agents can be geographically distributed but able to communicate via the system main blackboard.

Since the current implementation of the DYNACLIPS only uses CLIPS shell to represent an intelligent agent, we have chosen to follow the syntax of the CLIPS to represent facts, rules or commands. Knowledge Interface Format (KIF) [15,16] can also be used in the future to transfer facts, rules or commands.

REFERENCES

[1] Gilmore, J. F., Roth, S. P. and Tynor S. D. *A Blackboard System for Distributed Problem Solving.* Blackboard Architectures and Applications. Academic Press, San Diego, CA, 1989.

[2] Nii, H. P. *Blackboard Systems: The Blackboard Model of Problem Solving and the Evolution of Blackboard Architectures* , AI Magazine, Summer 1986, pp. 38-53.

[3] Nii, H. P. *Blackboard Systems: Blackboard Application Systems, Blackboard Systems from a Knowledge Engineering Perspective.* AI Magazine, August 1986, pp. 82-106.

[4] Diaz, A. C. and Orchard, R. A. *A Prototype Blackboard Shell using CLIPS.* Fifth International Conference on AI in Engineering, Boston, MA, July 1990.

[5] Orchard, R. A. and Diaz, A. C. *BB_CLIPS: Blackboard Extensions to CLIPS,* Proceeding of the First CLIPS User's Group Conference, Houston, TX, 1990.

[6] Dodhiawala, R. T., Sridharan, N. S. and Pickering C. *A Real-Time Blackboard Architecture,* Blackboard Architectures and Applications. Academic Press, San Diego, CA, 1989.

[7] Golstein, G. *PRAIS: Distributed, Real-Time Knowledge_Based Systems Made Eassy.* Proceeding of the First CLIPS User's Group Conference, Houston, TX, 1990.

[8] Myers, L. Johnson, C and Johnson, D. *MARBLE: A Systems for Executing Expert System in Parallel.* Proceeding of the First CLIPS User's Group Conference, Houston, TX, 1990.

[9] Schultz, R. D. and Stobie, I. C. *Building Distributed Rule-Based Systems Using the AI Bus.* Proceeding of the First CLIPS User's Group Conference, Houston, TX, 1990.

[10] Gallagher K. Q. and Corkill, D. D. *Performance Aspects of GBB,* Blackboard Architectures and Applications. Academic Press, San Diego, CA, 1989.

[11] Blackboard Technology Group, inc. *The Blackboard Problem Solving Approach,* AI Review, Summer 1991, pp. 37-32

[12] Gilmore, J. F., Roth, S. P. and Tynor S. D. *A Blackboard System for Distributed Problem Solving.* Blackboard Architectures and Applications. Academic Press, San Diego, CA, 1989.

[13] Rice, J., Aiello, N., and Nii, H. P. *See How They Run... The Architecture and Performance of Two Concurrent Blackboard Systems.* Blackboard Systems. Addison Wesley, Reading, Mass, 1988.

[14] Erman, L. D., Hayes-Roth, F., Lesser, R. V. and Reddy, D. R. *The Hearsay-II Speech-Understanding System : Integrating Knowledge to Resolve Uncertanity.* Blackboard Systems. Addison Wesley, Reading, Mass, 1988.

[15] Genesereth, M.R., Fikes, R. E. *Knowledge Interchange Format Reference Menual,* Stanford University Logic Group, 1992.

[16] Genesereth, M.R., Ketchpel, S. P., *Software Agents,* ACM Communication, July 1994, pp. 48-53.

[17] Cengeloglu, Y., Sidani, T. and Sidani, A. *Inter/Intra Communication in Intelligent Simulation and Training Systems (ISTS).* 14th Conference on Computers and Industrial Engineering, Cocoa Beach,March 1992.

[18] CLIPS Version 5.1 User's Guide, NASA Lyndon B. Johnson Space Center, Software Technology Branch, Houston, TX, 1991

A GENERIC ON-LINE DIAGNOSTIC SYSTEMS (GOLDS) TO INTEGRATE MULTIPLE DIAGNOSTIC TECHNIQUES

Giarratano, Boloor, Leibfried, Feagin, & Skapura

OMIT TO END

Abstract unavailable at time of publication.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	Nov/94	Conference Proceedings, September 1994

4. TITLE AND SUBTITLE

Third CLIPS Conference Proceedings - Volumes I and II

5. FUNDING NUMBERS

6. AUTHOR(S)

Gary Riley, editor

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Lyndon B. Johnson Space Center
Houston, Texas 77058
I-NET, Inc.
Houston, Texas 77058

8. PERFORMING ORGANIZATION REPORT NUMBERS

S-785

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

National Aeronautics and Space Administration
Washington, DC 20546-0001

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

NASA CP-10162

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT

Unclassified/Unlimited
Available from the NASA Center for AeroSpace Information
800 Elkridge Landing Road
Linthicum Heights, MD 21090-2934
(301) 621-0390

Subject Category: 61

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

Expert systems are computer programs which emulate human expertise in well defined problem domains. The potential payoff from expert systems is high: valuable expertise can be captured and preserved, repetitive and/or mundane tasks requiring human expertise can be automated, and uniformity can be applied in decision making processes. The C Language Integrated Production System (CLIPS) is an expert system building tool, developed at the Johnson Space Center, which provides a complete environment for the development and delivery of rule and/or object based expert systems. CLIPS was specifically designed to provide a low cost option for developing and deploying expert system applications across a wide range of hardware platforms. The development of CLIPS has helped to improve the ability to deliver expert system technology throughout the public and private sectors for a wide range of applications and diverse computing environments. The Third Conference on CLIPS provided a forum for CLIPS users to present and discuss papers relating to CLIPS applications, uses, and extensions.

14. SUBJECT TERMS

Expert Systems, Programming Languages, Computer Techniques

15. NUMBER OF PAGES

401

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	Unlimited